Universidad Autónoma de Madrid
Escuela politécnica superior

Double Degree in Mathematics and Computer Science

B.Sc. Thesis

FDA-PY: DEVELOPMENT OF A PYTHON PACKAGE FOR FUNCTIONAL DATA ANALYSIS

Author: Miguel Carbajo Berrocal
Tutor: Alberto Suárez González

May 2018

Universidad Autónoma de Madrid
Escuela politécnica superior

Doble Grado en Matemáticas e Ingeniería Informática

Trabajo Fin de Grado

FDA-PY: DESARROLLO DE UN PAQUETE PYTHON PARA EL ANÁLISIS DE DATOS FUNCIONALES

Autor: Miguel Carbajo Berrocal
Tutor: Alberto Suárez González

Mayo 2018

FDA-PY: DEVELOPMENT OF A PYTHON PACKAGE FOR FUNCTIONAL DATA ANALYSIS

Author: Miguel Carbajo Berrocal
Tutor: Alberto Suárez González

Dpto. de Ingeniería Informática
Escuela Politécnica Superior
Universidad Autónoma de Madrid
May 2018

Abstract

Resumen

El análisis de datos funcionales o FDA (*Functional Data Analysis*) por sus siglas en inglés, es un campo de la estadística que estudia datos en forma de funciones. Este tipo de análisis de datos tiene en cuenta que el proceso subyacente que genera los datos es un proceso estocástico y gracias a esto puede dar respuesta a preguntas que el análisis multivariante tradicional no es capaz.

Como FDA es un campo bastante reciente aún no está tan extendido como el análisis multivariante. Por esto mismo todavía existen pocas soluciones software que den soporte a las técnicas del este campo. Además aquellas que existen están en lenguajes de programación como MATLAB o R y no han sido construidas siguiendo esquemas de desarrollo *open-source* lo que dificulta que la comunidad contribuya a las mismas. Tampoco existe todavía ninguna solución en Python. Por estos motivos y además teniendo en cuenta que Python es el lenguaje elegido por gran cantidad de científicos y programadores, una implementación de las técnicas del FDA en este lenguaje parecía necesaria.

El análisis de datos funcionales es un campo muy extenso que además sigue creciendo gracias al trabajo de numerosos equipos en todo el mundo. Por este motivo el objetivo del proyecto no era simplemente crear una *toolbox* para FDA en Python, sino también dar comienzo a un proyecto *open-source* en el que cualquiera pueda contribuir. De esta manera conseguir que el paquete crezca y se mantenga la día con la actualidad del FDA una vez que este proyecto concluya.

Con estos objetivos en mente se dió comienzo al proyecto FDA. Actualmente consiste en un paquete Python publicado en un repositorio público en *GitHub* y aunque de momento sólo incluye funcionalidades básicas de FDA, aspira a largo plazo a convertirse en una herramienta completa y de uso extendido.

Palabras Clave

Análisis de datos funcionales, FDA, Python, software.

Abstract

Functional data analysis (FDA) is a field of Statistics in which the data considered are functions. This kind of data analysis takes into account that the underlying mechanism that generates the observations is a stochastic process. In this manner, it is possible to address some important questions (e.g. related to the continuity and smoothness of the functions measured) that traditional multivariate analysis can not.

Given that it is a relatively recent field of study, the methods of FDA are not as widely known as those of multivariate analysis. Specifically, there are few software implementations of FDA tools available, mostly in MATLAB and R. However, these are not built following open-source schemes, which makes it difficult to meke contributions to them. Up to this day there are not any packages for Functional Data Analysis in Python. Since Python is the language of choice for many data scientists and software developers, an implementation of FDA techniques in this programming language seems to be of some utility.

Functional data analysis is a vast field that keeps growing thanks to the work of numerous research teams around the world. That is why the aim of this project is not simply to design a FDA toolbox in Python but also to start an open-source project to which anybody can contribute. In this manner, the package can grow and be kept up to date even after the completion of this undergraduate thesis project.

With these goals in mind, the fda Python project has been published in a public repository in *GitHub*. At the moment it includes basic FDA functionalities. In the long term we expect that it will become a comprehensive and widely-used toolbox for FDA.

Key words

Functional data analysis, FDA, Python, software.

Special thanks

I would like to thank my tutor Alberto for proposing this very attractive project and for introducing me to the field of functional data analysis. Special thanks to Carlos Ramos that has been very involved in the project and that has shown me so much about Python. Without his help the project would not be what it is today.

Also a big thank you for all my friends and family that gave me all their support and had to stand my bad mood when everything was not going as planned. I know I have been declining and postponing a lot of amazing plans because I was busy working but I promise I will make it up you.

Contents

Figures Index

1 Introduction

Functional data analysis (FDA) is a branch of Statistics that studies data in the form of functions. Initial contributions in this area appear in the 1950's in publications such as Fehahder U. *Stochastic processes and statistical inference* [1] and Rao C.R. *Some statistical methods for comparison of growth curves* [2]. However, the term FDA is not used until 1982 by J.O. Ramsay in the article *When the data are functions* [3]. Since then numerous articles and reports have been published in the area of functional data analysis. Nowadays, it is still an important research field in Statistics.

Numerous software packages and libraries in different programming languages have been developed to implement the statistical techniques of FDA. Amongst them there are the *refund* [4] and *fda.usc* [5] packages for R and the PACE package by the Statistics Department from University of California for MATLAB [6]; finally, the package fda [7], which is available in both languages. However, there is no Python package that provides a comprehensive implementation of the techniques of Functional Data Analysis.

Python is one of the fastest growing programming languages in both the academic and business worlds. It has been ranked by IEEE as the topmost programming language in 2017 [8]. In addition, it is one of the most widely used languages in data science. Python boasts also a large community of contributors committed to the development and maintenance of scientific software. For these reasons, the design and implementation of a package in Python for Functional Data Analysis seems to be a worthwhile endeavour. The availability of such a tool would facilitate the development of a large number of applications and projects that involve the analysis of functional data.

1.1 Goals and scope

The goal of this work is to initiate an open-source project to develop a Python package that provides support to functional data analysis. Two R packages that implement FDA methods have been used as references for implementation: J.O. Ramsay et al. *fda: Functional Data Analysis*, 2017 [9] and M. Febrero-Bande et al. *Statistical computing in functional data analysis: The R package fda.usc* [10].

Given the large extent of this field and the size of these packages, the scope of this under-

graduate thesis project is limited. The ultimate goal is to provide the Python community with a complete tool. This is the main reason why the project is released in an open-source formant, so that it can be enriched with contributions from others. Given the nature of the project, it is important to follow a rigorous methodology during the construction of the software. Special attention has been devoted to aspects such as documentation, testing, and continuous integration.

Since our aim was to produce the most complete package we could, given the time constraints of an undergraduate thesis project, the complete scope of the project was not clearly delimited at the beginning. For this reason, an iterative and incremental methodology was used to develop the software package. The tasks to be completed in each cycle were defined during regular meetings, in most cases, with a weekly periodicity, at the beginning of each iteration.

1.2 Document structure

Besides the introduction, the document is composed of three chapters and three appendices. In Chapter 2 the theoretical framework for the different functionalities developed in the fda Python package is given in some detail. Specific algorithms used in the implementation of these functionalities are also presented. For the sake of clarity, some of the derivations and descriptions of algorithms have been included in appendix A. The figures included were generated using our fda Python package developed in this project. The scripts to generate these figures are included in Appendix B.

In Chapter 3 we discuss the methodology used in the project. In the first section of this chapter a preliminary analysis and the specification of the software requirements are presented. In Section 3.2 the design of the package is presented in detail. Connections between the design and the theoretical derivations given in Chapter 2 are also detailed. The question of how to ensure the quality of the software in an open-source project is addressed in the final part of this chapter. Finally, the tools and workflow employed to achieve this purpose are discussed.

To conclude, Chapter 4 summarises the contributions made and outlines the long-term goals and future steps in the project initiated in this work.

Appendix A includes equations and algorithms, which were not included in Chapter 2. As discussed earlier, the scripts used to generate the figures included in this document are collected in appendix B. The documentation of the fda package is provided in Appendix C. This material is structured as a separate document with its own pagination.

2

Functional Data Analysis

Assume that we are studying the growth of a population. For this purpose we have taken a series of measurements of the heights for each subject in the studied population at different ages. Some questions on these data, such as 'How does a person grow on average?' or 'How fast does a person grow at a certain age?' can be considered. Standard multivariate data analysis can be used to answer only some of these questions. Others, such as the rate at which a person grows, can only be addressed when the functional nature of the data is explicitly considered.

Functional data analysis studies these types of scenarios when there is an underlying function to the data studied. It takes advantage and also tries to explain properties related to the functional nature of the data. For example studying derivatives or making assumptions that functions are generally continuous and smooth, in some sense.

FDA can be more formally defined as the branch of statistics that studies data that are realisations of a stochastic process.

2.1 Functional data representation

As discussed earlier, a functional datum is a realisation of a stochastic process. Since functions depend on a continuous parameter (e.g. time or space), it is impossible to have a representation that consists of measurents at each point of the function's domain. A first approach to storing functional data is to record the values of the function in a finite grid of points. This is referred to as the **discrete representation** of a functional datum and will be discussed in Section 2.1.1.

In some applications, a representation based on discrete measurements is insufficient. To work with actual functions, a basis representation can be used instead. In particular, there are large classes of functions can be represented as linear combinations of monomials or of sinusoidal functions. Monomials or sinusoidal functions of different frequencies form bases that span infinite-dimensional functional spaces. However, most functions need to be represented an infinite number of basis functions (e.g. the Taylor series representation $e^x = \sum_{n=0}^{\infty}$). When these types of representations are restricted so that they include a limited number of basis functions one obtains finite dimensional subspaces of the original functional space. A function in such a subspace can actually be stored as the coefficients of the linear combination (the *coordinates*). Since the number of basis functions is finite, the number of coordinates needed is also finite. The

representation of functional data as a linear combination of basis functions will be discussed in Section 2.1.2.

There is not a single representation that is best for all cases. Specifically some operations are simpler in one representation than in the other. The choice of representation is determined by the type of analysis that needs to be carried out.

2.1.1 Discrete representation

A single functional datum consists in multiple observations taken over a continuum that jointly form a curve, a surface or any quantity that can be represented as a (possibly multivariate) function. For simplicity we consider one-dimensional functions $x(t)$, with t in some finite continuous domain. Let $x(t_j)$ be the value of the function at the observation point t_j. The value registered y_j, which could be different from $x(t_j)$; e.g. because of measurement noise.

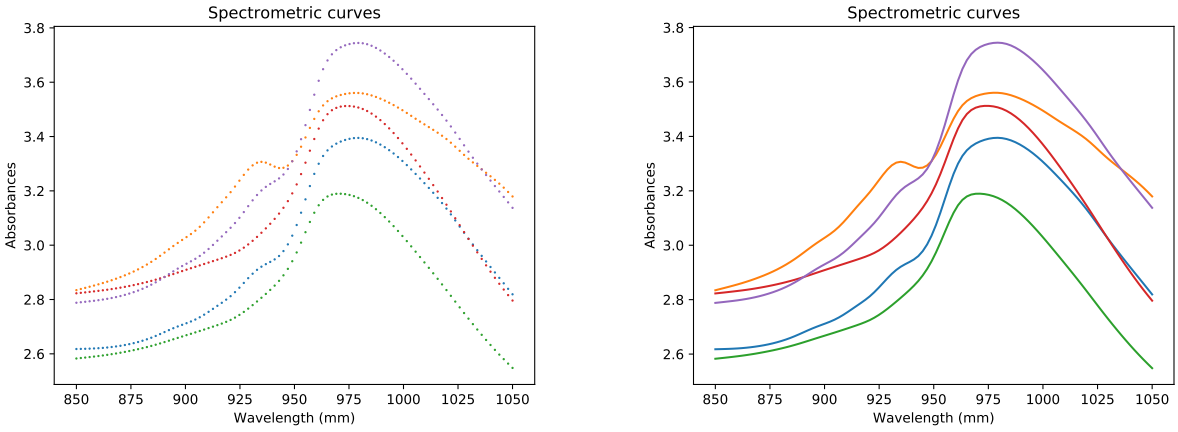
In most cases we will have at our disposal different samples $\{x_1(t), x_2(t), \ldots\}$, each of which is a realisation of the underlying process.

For example, in an experiment conducted to study the growth in humans each height measurement corresponds to an observation and every subject in the experiment is a sample.

Typically we will work with sets of n samples, each of which consists in m observations: $\{y_{ij}\}$ for $i = 1, ..., n$ $j = 1, ..., m$. The first index i refers to samples. The second one, j, is used to label to the observations. Hence, the value y_{ij} is the j-th observation in the i-th sample.

Figure 2.1a is the graphic representation of 5 functional data from the Tecator dataset [11]. Each of this curves corresponds to the dependence of the infrared absorbance as a function of the frequency of the radiation for different meat samples.

Despite being stored in its discrete representation, these functional data are displayed in figure 2.1b as continuous curves using linear interpolation.



(a) Discrete representation of functional data.

(b) Usual representation of functional data.

Figure 2.1: Tecator dataset.

2.1.2 Basis representation

Given a functional space $\mathcal{E}$, a basis function system is defined as a set of functions $\{\phi_k\}_{k=1,2,...}$ whose linear combination can represent any function defined in $\mathcal{E}$. A very simple basis system is the one formed by all monomials $1, t, t^2, ..., t^k, ...$

Many functional spaces are infinite dimensional. Hence, the number of elements of the corresponding basis systems is infinite. In practical applications, we work in finite dimensional subspaces by considering finite linear combinations of basis functions of the form

$$x(t) = \sum_{k=1}^{K} c_k \phi_k(t) \tag{2.1}$$

where $\boldsymbol{c} = (c_1, c_2, ..., c_K)$ is a vector of coefficients and $\phi_1, \phi_2, ..., \phi_K$ a finite set of basis functions.

In the case of the monomial basis the expansion would look like:

$$x(t) = c_0 + c_1 t + c_2 t^2 + ... + c_K t^K$$

There are several function approximations that can be represented using a basis of monomials. For instance, any continuous function in $\mathcal{L}^2([a, b])$ can be represented using Bernstein polynomials. A K-differentiable function can also be locally represented by its Taylor polynomial of order K using the monomial basis system.

For some applications, monomial basis systems work surprisingly well given their simplicity. However, other types of functions have a more natural representation in other kinds of basis systems. For instance, periodic functions can be represented as **Fourier series**. The basis of **B-Splines** is used to approximate smooth functions using piecewise polynomials.

Fourier series

The orthogonal Fourier basis consist in a constant function and a set of sines and cosines of different frequencies:

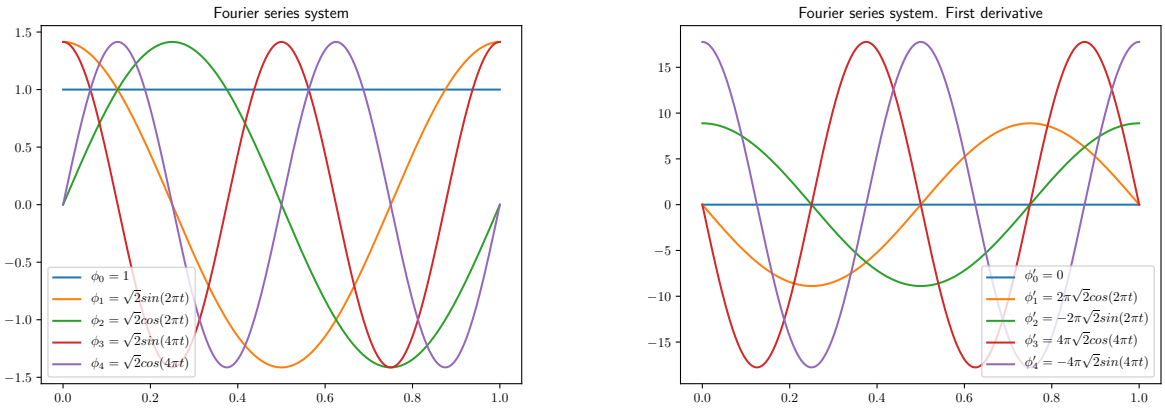
$$\begin{aligned} \phi_0(t) &= \frac{1}{\sqrt{2}} \\ \phi_{2n-1}(t) &= \sin\left(\frac{2\pi n}{T} t\right) \\ \phi_{2n}(t) &= \cos\left(\frac{2\pi n}{T} t\right) \end{aligned} \tag{2.2}$$

In considered in $\mathcal{L}_2[0, T]$, the orthonormal basis is

$$\begin{aligned} \phi_0(t) &= \frac{1}{\sqrt{T}} \\ \phi_{2n-1}(t) &= \sqrt{\frac{2}{T}} \sin\left(\frac{2\pi n}{T} t\right) \\ \phi_{2n}(t) &= \sqrt{\frac{2}{T}} \cos\left(\frac{2\pi n}{T} t\right) \end{aligned} \tag{2.3}$$

Fourier series system work well when the process generating the data is periodic and when it is stable, meaning the curvature stays the same order all throughout the domain [12].

Other advantages of the Fourier series is that differentiation of a function represented in this basis can be done analytically. Furthermore, the derivative can also be represented in a Fourier series system.



(a) Basis.

(b) First derivative.

Figure 2.2: Fourier basis. Number of basis functions = 5.

B-splines

Basis splines or B-splines are a common choice of basis system when the data analysed are not periodic. Splines are piecewise polynomials that connect a series of points, which are called **knots** or **breakpoints**. Because they are polynomials, they allow fast computations. However, they are more flexible than standard polynomials. The monomial basis system, which was used as an example at the beginning of the Section 2.1.2, can be seen as a particular case of these basis systems.

Before defining the basis, we need to give a formal definition of a spline. A spline of order m with knots $t_0, t_1, ..., t_{l-1} \in [a, b] \subset \mathbb{R}$, $t_0 \leq t_1 \leq ... \leq t_{l-1}$ is a real function $f : [a, b] \subset \mathbb{R} \longmapsto \mathbb{R}$ that satisfies:

- in each interval $[t_i, t_{i+1}]$ f is a polynomial of degree at most $m - 1$.
- $f \subset C^{m-2}((a, b))$, f can be differentiated up to $m - 2$ times over (a, b). [13] [12]

The latter condition means that when two polynomial pieces meet at a knot their values and up to the $m - 2$-th derivatives are equal. For instance, an order 2 spline is continuous at the knots; an order 3 spline is continuous differentiable at the knots; an order 4 spline is continuous and has two continues derivatives at the knots, and so on.

Given a sequence of knots $\tau = \{t_i\}_{i=0,1,...,l-1}$ and an order m, all splines of order m and knots τ define a vector space. Among all the possibilities for choosing a base for this space, the most used one is B-splines, a basis system created by de Boor (1978). [14].

The number of basis functions to define the B-splines system is its order plus the number of knots minus two [12]. A spline of order m and knots τ is defined as:

$$S(t) = \sum_{k=1}^{m+length(\tau)-2} c_k B_{k,m,\tau}(t) \tag{2.4}$$

where $B_{k,m,\tau}(t)$ are the spline basis functions defined recursively as:

$$
\begin{aligned}
B_{k,1,\tau}(t) &= \begin{cases} 1 & if \quad t_k \leq t < t_{k+1}, \\ 0 & elsewhere \end{cases} \\
B_{k,l,\tau}(t) &= \frac{t - t_k}{t_{k+l} - t_k} B_{k,l-1,\tau}(t) + \frac{t_{k+l+1} - t}{t_{k+l+1} - t_{k+1}} B_{k+1,l-1,\tau}(t)
\end{aligned}
\tag{2.5}
$$

We have and example of B-splines in Figure 2.3. This basis system was built for order 4 splines (cubic splines) and a sequence of 7 knots equally spaced between 0 and 1. Note that is not necessary to place the knots at the equal distances, as in the example. In general, more knots should be placed where the function has a stronger variation, so that the splines representation is more flexible in the vicinity of those points [12].

A careful examination of Equation (2.5) and of the plots Figure 2.3 reveals that B-splines are positive over no more than m intervals. This property is called **compact support**. It is essential to the efficient computation of a B-spline system [12].

Also note the behaviour of B-splines close to the boundaries in figure 2.3. The furthest left spline is discontinuous at 0. The following spline is only continuous but not differentiable. The following spline is differentiable twice. The same pattern is observed at the right boundary. These discontinuities allow the splines generated by the basis system to be irregular outside the interval considered. This behaviour is achieved by placing m knots at the same location on the boundaries [12].

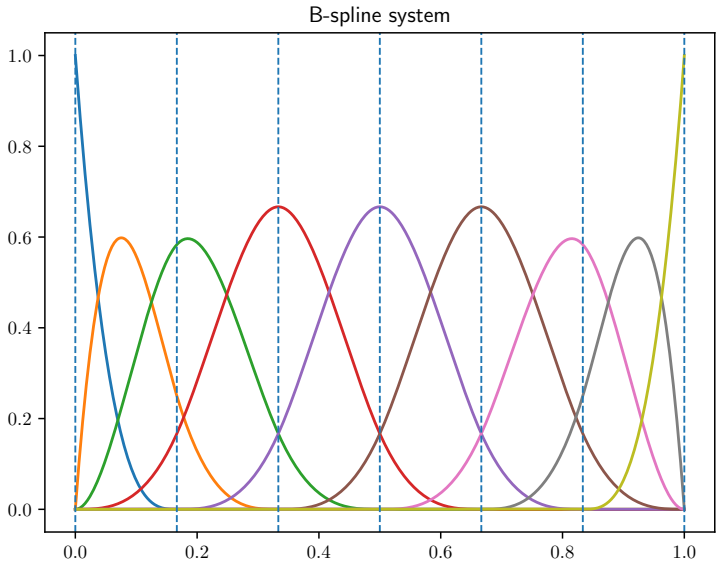


Figure 2.3: Order 4 B-Spline (cubic splines). Seven knots equally spaced between 0 and 1. The vertical dashed lines indicate where the knots are placed.

2.2 Working with discrete functional data

In this section we will discuss the theoretic framework and provide illustrations of the functionality implemented for the discrete representation of functional data in our Python package. This section covers the computation of basic statistics, differentiation, kernel smoothing and principal component analysis in FDA.

2.2.1 Basic statistics

One of the first analysis one should make when working with a new dataset is to compute simple yet useful statistics such as the mean, the variance and autococovariances.

Let $\{x_i(t)\}_{i=1,...,n}$ a set of functional data defined in the same functional space. The following statistics are defined:

Mean

$$\bar{x}(t) = \frac{1}{n}\sum_{i=1}^{n} x_i(t) \tag{2.6}$$

Variance

$$\sigma^2(t) = \frac{1}{n}\sum_{i=1}^{n} (x_i(t) - \bar{x}(t))^2 \tag{2.7}$$

Autocovariances

$$\Sigma(s,t) = \frac{1}{n}\sum_{i=1}^{n} (x_i(s) - \bar{x}(s))\,(x_i(t) - \bar{x}(t)) \tag{2.8}$$

Geometric mean

$$\left(\prod_{i=1}^{n} x_i(t)\right)^{\frac{1}{n}} \tag{2.9}$$

When we are working with discrete functional data the computation of these statistics is actually equivalent to computing these corresponding multivariate statistics assuming that the set of measurements taken in the same sampling point is an independent random variable.

As a reminder, a functional data discrete dataset is set of measurements $\{y_{ij}\}_{\substack{i=1,...,n \\ j=1,...,m}}$ and a set of sampling points $\{t_j\}_{j=1,...,m}$ where each subset $\{y_{ij}\}_{j=1,...,m}$ consists in the measurements taken at each sampling point of a same process and together consist in a single functional datum. Thus, the basic statistics are computed as follows:

- Mean: $\bar{y}_j = \frac{1}{n}\sum_{i=1}^{n} y_{ij} \quad j = 1, 2, ..., m$
- Variance: $\sigma_j^2 = \frac{1}{n}\sum_{i=1}^{n} (y_{ij} - \bar{y}_j)^2 \quad j = 1, 2, ..., m$
- Covariance: $\Sigma_{jk} = \frac{1}{n}\sum_{i=1}^{n} (y_{ij} - \bar{y}_j)\,(y_{ik} - \bar{y}_k) \quad j, k = 1, 2, ..., m$
- Geometric mean: $(\prod_{i=1}^{n} y_{ij})^{\frac{1}{n}} \quad j = 1, 2, ..., m$

Although the geometric mean is usually defined as above their equivalent expression

$$\exp\left(\frac{1}{n}\sum_{i=1}^{n} ln(y_{ij})\right)$$

is preferred as it avoids the arithmetic overflow problems that the first formula can easily have. Basic mathematical libraries as *scipy* use this last expression for its implementation.

2.2.2 Differentiation

One of the advantages of FDA over traditional multivariate analysis is that by treating data as functions we are able to study their derivatives. Often, the key to understanding the underlying process that generates the data is in their variation. For this reason, differentiation plays a crucial role in FDA.

Since our data are not actual functions, we have to work with approximations of the derivative for discrete data. The algorithm chosen for this purpose its the lagged differences estimate. To compute the derivative at a certain instant t_j one takes the slope of the line resulting of the linear interpolation between the data for (t_{j-1}, y_{j-1}) and (t_{j+1}, y_{j+1}).

$$D_{ij}^1 = \begin{cases} \frac{y_{i1}-y_{i2}}{t_1-t_2} & \text{if } j = 1 \\ \frac{D_{i(j-1)}-D_{i(j+1)}}{t_{j-1}-t_{j+1}} & \text{if } 1 < j < m \\ \frac{y_{i(m-1)}-y_{im}}{t_{m-1}-t_m} & \text{if } j = m \end{cases} \tag{2.10}$$

For computing higher order derivatives the algorithm is used iteratively.

2.2.3 Smoothing

Usually when working with functional data one of the first assumptions made is that the underlying process of the studied data is smooth [12]. Smoothness is a vague concept, which can be understood in different ways. Nonetheless, the goal is to reduce the abrupt variations that our noisy samples have, regardless of whether we use some smoothness criterion based on derivatives or simply perform a linear interpolation.

Although there are different ways of smoothing a function, we will consider only methods based on regression [15]. We assume that the functional observations are the combination of an underlying smooth process and an error (sometimes called noise) that has a certain distribution with mean 0.

$$y_j = x(t_j) + \epsilon_j, \qquad \mathbb{E}(\epsilon_j) = 0 \quad j = 1, 2, ..., m \tag{2.11}$$

Our goal is to estimate the function $x(t)$ or at least $x(t_1), x(t_2), ..., x(t_m)$. The estimation is denoted $\hat{y}(t)$ and its values at $t_1, t_2, ..., t_m$ are referred as $\hat{y}_1, \hat{y}_2, ..., \hat{y}_m$ respectively.

2.2.4 Kernel smoothers

In the particular case of these types of smoothers, kernels are used to identify the underlying process and filter the noise out. A **kernel** is any smooth function that meets the following criteria:

$$\int K(t)dt = 1, \qquad \int tK(t)dt = 0 \qquad \text{and} \qquad \sigma_K^2 \equiv \int t^2 K(t)dt > 0 \tag{2.12}$$

Examples of kernels (see figure 2.4) are the normal distribution,

$$K(t) = \frac{1}{\sqrt{2\pi}} \exp\left\{-\frac{1}{2}t^2\right\}, \tag{2.13}$$

the Epanechnikov kernel

$$K(t) = \begin{cases} 0.75(1-t^2) & \text{if } |t| \leq 1 \\ 0 & \text{elsewhere} \end{cases} \tag{2.14}$$

or the uniform kernel

$$K(t) = \begin{cases} 0.5 & \text{if } |t| \leq 1 \\ 0 & \text{elsewhere} \end{cases} \tag{2.15}$$

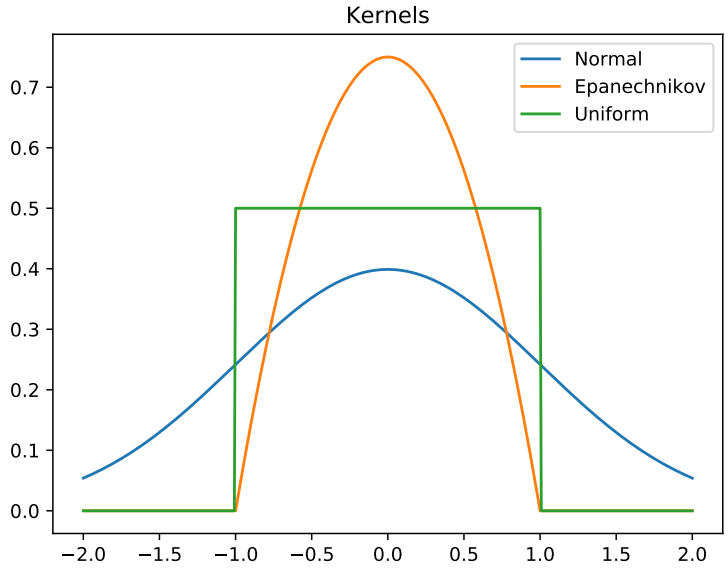


Figure 2.4: Normal kernel in blue, Epanechnikov kernel in orange and uniform kernel in green.

By using kernels we are able to estimate the value of $x(t)$ using weighted averages of the neighbouring observations to instant t.

The smoothers included in the fda Python package are cases of linear smoothers. An estimator $\hat{y}$ of x is called a **linear smoother** if, for each t, there exists a vector $l(t) = (l_1(t), l_2(t), ..., l_m(t))'$ such that

$$\hat{y}_i(t) = \sum_{i=1}^{m} l_i(t) y_i \tag{2.16}$$

Then, we can define the vector of fitted values

$$\hat{\boldsymbol{Y}} = (\hat{y}_1, \hat{y}_2, ..., \hat{y}_m) \equiv (\hat{y}(t_1), \hat{y}(t_2), ..., \hat{y}(t_m))$$

as the matrix product of a matrix $\hat{H}$ and the vector of observations $Y = (y_1, y_2, ..., y_n)$

$$\hat{Y} = \hat{H}Y \tag{2.17}$$

where $\hat{H}$ is a $m \times m$ dimensional matrix and $\hat{H}_{ij} = l_j(t_i)$. This matrix is called **smoothing matrix** or **hat matrix**.

One of the kernel smoothers implemented in the package is the **Nadaraya-Watson kernel smoother**. As a linear smoother, it is of the form given in Equation (2.16)

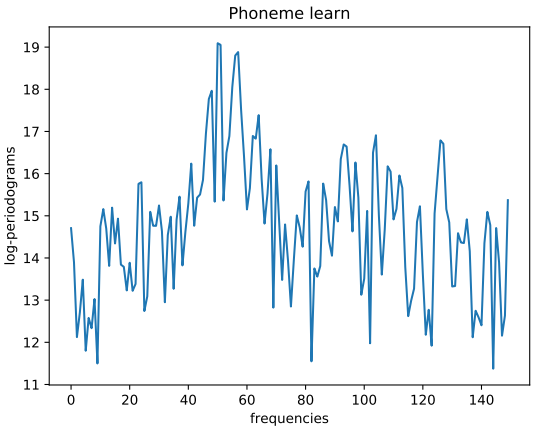
$$\hat{y}(t) = \sum_{i=1}^{m} l_i(t) y_i$$

where

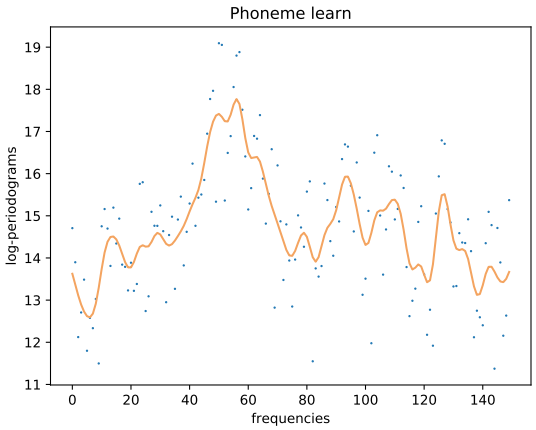
$$l_j(t) = \frac{K\left(\frac{t-t_j}{h}\right)}{\sum_{k=1}^{m} K\left(\frac{t-t_k}{h}\right)} \tag{2.18}$$

for $h > 0$ where K is a kernel function as stated in Equation (2.12).

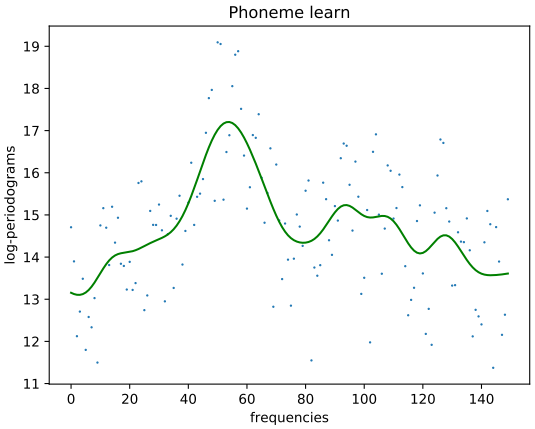
The bandwith of the kernel h is the **smoothing parameter**. It is crucial to use an appropriate value of this parameter to obtain reasonable smoothed approximations. On the one hand, if the choice of h is too low the amount smoothing is insufficient and the data would still exhibit abrupt variations. On the other hand, if the value of h is too large, relevant features in the data would be wiped out. Figure 2.5 illustrates different choices of the smoothing parameter.



(a) Original data



(b) Under-smoothed data



(c) Well smoothed data

(d) Over-smoothed data

Figure 2.5: Smoothing using Nadaraya-Watson with a normal kernel using different bandwidths. $h = 2$ top-right. $h = 5$ bottom-left. $h = 15$ bottom-right.

The question is now how to determine a good value of the smoothing parameter, one that avoids undersmoothing or oversmoothing the data. There are several methods for accomplishing this task: cross-validation (CV), general cross-validation (GCV) and other general bandwidth selection criteria. In this part of the document we describe GCV, which is the parameter selection

technique used in the examples. Details of the the methods implemented are given in Section A.3 of the appendix.

The **general cross validation score (GCV)** is

$$GCV(h) = \frac{1}{n}\sum_{j=1}^{m}\left(\frac{y_j - \hat{y}_j}{1 - tr(\hat{H})/m}\right)^2 \tag{2.19}$$

where the hat matrix $\hat{H}$ depends on the parameter h.

The lower the value of GCV, the better the smoothing is. Hence, the goal is to **minimise** the score. However, $GCV(h)$ does not necessarily have a minimum. In consequence, it is a good idea to analyse its variation as a function of h.

In general, we have more than one sample at our disposal. We will assume that the error has the same distribution for all the samples. If this is the case, it is appropriate to use the same value of the smoothing parameter for all samples. What we do in this case is to add up all the GCV scores of the different samples for each choice of h and then select the one that minimises the sum.

To illustrate how to work with kernel smoothers we have replicated an example that can be found in the publication of M. Febrero-Bande and M. Oviedo in the *Journal of Statistical Software* [5] for which they used their own R library *fda.usc*.

Figure 2.6 shows the first five samples of the phoneme dataset [16] that contains data from 250 speech frames and for each frame it has log-periodograms for 150 observations. As it can be seen in the figure, these data are very noisy and as the data is, the error may complicate the study of the underlying process if not taken care of. It is a very good example to see how useful is to apply smoothing to obtain 'clean' patterns out of the data and get rid of the noise.

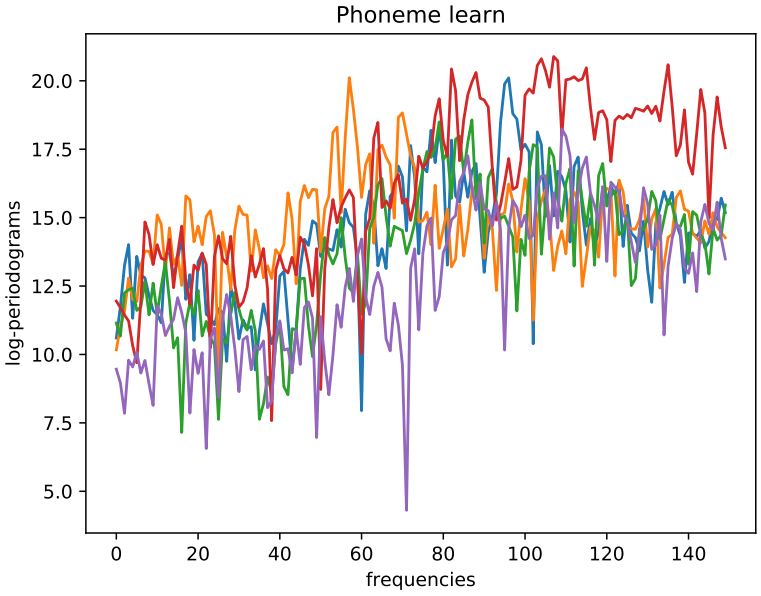


Figure 2.6: Phoneme dataset subset raw data.

We start by calculating which parameter is best for each of the three kernel smoothers implemented in the package (Nadaraya-Watson, local linear regression and KNN) using the GCV score. We can see the scores obtained for different choices of parameter in figure 2.7. Now, we only need to pick the one that minimises this score.

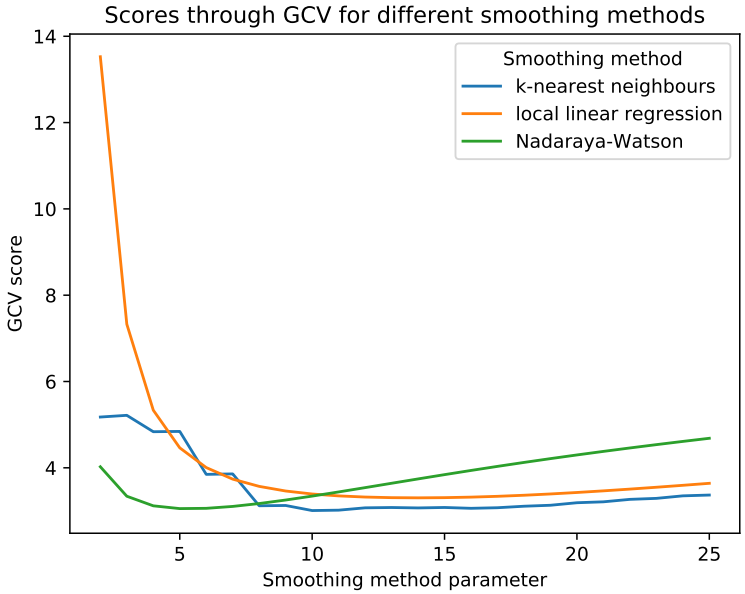
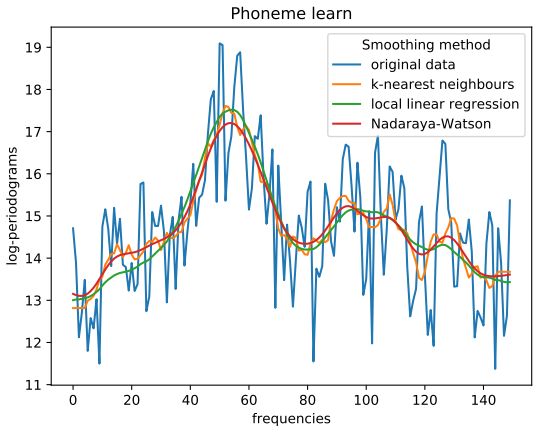
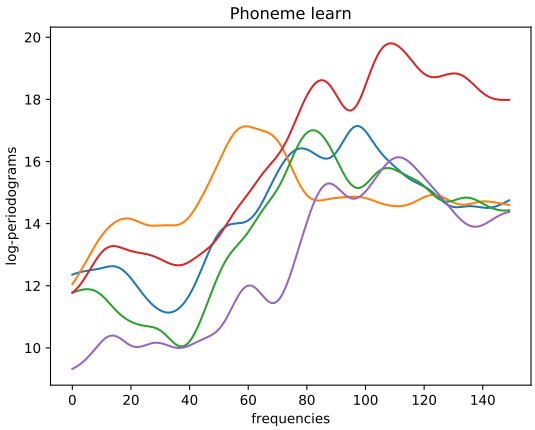


Figure 2.7: GCV scores for the three different smoothing methods.

In figure 2.8a we can see how the different methods smooth the 11th sample of the dataset. Figure 2.8b shows the same 5 first samples that were shown at the beginning of the example once they have been smoothed using the Nadaraya-Watson smoother with the best parameter choice.



(a) Eleventh sample of the phoneme dataset. Original and smoothed by 3 different methods.



(b) Phoneme dataset subset smoothed data.

Figure 2.8: Smoothing results.

2.2.5 Functional principal components analysis

We often want to understand how different the functional data in a sample are, and more precisely where or in what they differ. Covariance and correlation functions provide this information. However, they are often difficult to interpret and visualise. For multivariate data, **Principal component analysis (PCA)** is used to obtain the main modes of variation across the data set.

textbfFunctional principal components analysis (FPCA) is the extension of PCA to functional data.

The first principal component is the projection of the functional data that has the largest possible variance. Subsequent principal components are found by finding projections that capture the largest part of variance across the sample and are orthogonal to the previous ones.

The details of the algorithm are as follows: The first iteration consists in finding the projection function $\boldsymbol{\xi}_1(s)$ that maximises (across data variance)

$$\frac{1}{n}\sum_{i=1}^{n}\left(\int \boldsymbol{\xi}_1(s)x_i(s)ds\right)^2 \tag{2.20}$$

under the constraint $||\boldsymbol{\xi}_1||^2 = 1$ in the sense of the $\mathcal{L}_2$ norm.

The following iterations consist in finding orthonormal projection functions $\boldsymbol{\xi}_2, \boldsymbol{\xi}_3, ...$ maximising Equation (2.20) subject to the constraints to ensure orthonormality. In iteration p, $\boldsymbol{\xi}_p$ has to maximise the above formula under the normality constraint $||\boldsymbol{\xi}_p||^2 = 1$ and the ones below assuring the orthogonality with the previous $p-1$ projection functions:

$$\int \boldsymbol{\xi}_p(s)\boldsymbol{\xi}_k(s)ds = 0, \qquad \forall k < p \tag{2.21}$$

At the end we will have obtained an orthonormal functional basis $\{\boldsymbol{\xi}_i\}_{i=1,2,...}$ in which we can represent our functional data. The amount of variance across the data explained by each of the components decreases with each component. This is very useful because it allows to understand how the samples differ from one another simply by studying a few first principal components. For example, if the values of the first component $\boldsymbol{\xi}_1$ are very high or very low over an interval of the domain $[a, b]$ this means that just by studying the data over that interval we will find the greatest differences among them.

FPCA is also very useful for clustering and dimensionality reduction purposes. One of the most common usages of FPCA is to project the data into the first n components. This way we are reducing the dimensions of the data we are working with with a very little loss of variation across the data. Samples that were very different in the original basis are still very different in the new projection. This is very useful for clustering algorithms because we are reducing the amount of data we have to process with no or very little consequences on the final results.

Computing FPCA with discrete functional data

When working with discrete data, the integrals in the previous formulae become sums of the discrete data in each sample and calculating weight vectors $\vec{\boldsymbol{\xi}_1}, \vec{\boldsymbol{\xi}_2}, ..., \vec{\boldsymbol{\xi}_n}$ as the discrete representation of the weight functions. Notice that we can only compute principal components up to the number of sampling points. The algorithm for computing FPCA in this case is equivalent to traditional PCA in the multivariate case, where every set of measurements taken in the same sampling point $\{x_{ij}\}_{i=1,..n}$ are treated as an independent variable.

Instead of proceeding with the iterative algorithm, more efficient ways of computing principal componets exist. The algorithm used in the fda Python package is the eigenvalue decomposition of the covariance matrix through singular value decomposition (SVD). It uses the efficient SVD implementation in the mathematics Python package *scipy*. Using this method we efficiently obtain not only the principal components $\vec{\boldsymbol{\xi}_1}, \vec{\boldsymbol{\xi}_2}, ..., \vec{\boldsymbol{\xi}_n}$, but also the coordinates of each functional datum in the basis generated by the principal components. It also provides a projection matrix to easily get the coordinates in the principal components basis of any new functional datum given in the original coordinate system.

This method of calculating the FPCA can be also used when working with functional data in a functional basis representation by first evaluating them in a grid fine enough to capture its features. [12]

2.3 Working with functional data in a basis system

As seen in the previous section Functional data are usually recorded as a discrete set of measurements. However, in some applications one would like to work with structures for representing these data that are closer to their functional nature. What we have in mind is the basis representation previously discussed in Section 2.1.2. Actually, most of work done in the FDA literature focuses on this latter representation. It is not uncommon to find the suggestion that the discretly observed data should be transformed and represented in a linear combination of basis functions, which would then constitute the object of analysis [17]. We will start this section about talking this transformation of data from one representation to the other. It will be followed by a section talking about how to add smoothing to this process and finally definition and computation of basic statics will be discussed.

2.3.1 From discrete data to a functional basis system

Remember that we have a set of measurements $\{y_{ij}\}_{\substack{i=1,...,n \\ j=1,...,m}}$ taken in a series of points $\{t_j\}_{j=1,...,m}$. For simplicity we will work with just one sample $\boldsymbol{y} = \{y_j\}_{j=1,...,m}$ but the reader will realise that the problem can be easily generalised by switching the vector of $\boldsymbol{y}$ of observations for a matrix.

Given a basis function system of K basis functions $\phi = \{\phi_k\}_{k=1,...,K}$ we want to get closest representation of the vector $\boldsymbol{y}$ in the function space generated by ϕ. The solution has to be a linear combination of the basis functions:

$$\hat{y}(t) = \sum_{k=1}^{K} c_k \phi_k(t) = \boldsymbol{c}'\phi(t) \tag{2.22}$$

where $\boldsymbol{c} = (c_1, c_2, ..., c_K)$ is the coefficient vector we will need to calculate.

For every sampling point t_j we have that the sampling value is the value given by the model plus an error.

$$y_j = \hat{y}(t_j) + \epsilon_j \overset{\text{Notation}}{\equiv} \hat{y}_j + \epsilon_j \tag{2.23}$$

Our goal is to minimise the errors. There are several types of error estimates that can be used. The most widely used one in the sum of squared errors.

$$SSE(\boldsymbol{y}, \boldsymbol{c}) = \sum_{j=1}^{m} (\epsilon_j)^2 = \sum_{j=1}^{m} (y_j - \hat{y}_j)^2 \tag{2.24}$$

where the value of $\hat{y}_j$ depends on the choice of $\boldsymbol{c}$.

Equation (2.24) can be rewritten using matrix notation as

$$SSE(\boldsymbol{y}, \boldsymbol{c}) = (\boldsymbol{y} - \hat{\boldsymbol{y}})'(\boldsymbol{y} - \hat{\boldsymbol{y}}) \equiv \left(\boldsymbol{y} - \boldsymbol{c}'\Phi\right)'\left(\boldsymbol{y} - \boldsymbol{c}'\Phi\right) \tag{2.25}$$

where Φ is a $K \times m$ matrix whose columns are the basis functions evaluated at the sampling points $t_1, t_2, ..., t_m$.

Differentiating expression 2.25 with respect to $\boldsymbol{c}$ and equalling to 0 we obtain the following equation.

$$2\Phi\Phi'\boldsymbol{c} - 2\Phi'\boldsymbol{y} = 0$$

Solving the equation above for $\boldsymbol{c}$ we obtain the closed-form for the coefficient vector minimising 2.24.

$$\hat{\boldsymbol{c}} = \left(\Phi'\Phi\right)^{-1}\Phi'\boldsymbol{y} \tag{2.26}$$

When the errors are independent and identically distributed this model work fine but they do not, we can use all the information we can get about the distribution of errors (variances and covariances) to make a better fit.

Given a symmetric square matrix W of dimension the number of samples m noted weight matrix, we define the weighted sum of squared errors

$$SSE_W(\boldsymbol{y}, \boldsymbol{c}) = \left(\boldsymbol{y} - \boldsymbol{c}'\Phi\right)' W \left(\boldsymbol{y} - \boldsymbol{c}'\Phi\right). \tag{2.27}$$

If the variance-covariance matrix of the errors Σ_ϵ is available, a good choice for W is Σ_ϵ^{-1} [12].

As we did in the unweighted case, by differentiating and solving $\boldsymbol{c}$ we get the following closed formula for the coefficient vector minimising 2.27

$$\hat{\boldsymbol{c}} = \left(\Phi' W \Phi\right)^{-1}\Phi' W \boldsymbol{y}. \tag{2.28}$$

How many basis functions?

One question that arises is *How many basis functions should my basis system have?*. By incrementing the number of basis functions we increase the flexibility of the basis system allowing it to adjust better to the data. This might seen always as an advantage. However, it is not necessarily so. We are back to a problem similar to finding the appropriate balance between undersmoothing and oversmoothing, which was discussed in Section 2.2.4. On the one hand, selecting a number of basis functions that is too low can cause the loss of information on the features of the function. On the other hand, a number of basis functions that is too large, can lead to overfitting. This can introduce spurious features in the data representation. As we can observe in figure 2.9 for this particular functional datum, a B-spline basis of size 7 is not sufficient to capture the features in the original data. By contrast a basis of size 30 is too large. In consequence, the representation of the functional datum exhibits variations, which are probably an artifact. In this case, by using an intermediate number of basis functions, 15, we get a more reasonable (smooth) representation of the data.

2.3.2 Smoothing with roughness penalization

As we have seen in the previous Section 2.3.1 by representing data in a functional basis system we are already smoothing the data. Unfortunately, one does not have direct control on the type of smoothing that is being carried out. Furthermore, for some applications, one needs not only smooth function, but also functions with smooth derivatives.

Smoothing is understood as the process of reducing the **roughness** of a function. One of the most common definitions of roughness of a function $x(t)$ is the integral of its curvature

$$PEN_2(x) = \int D^2 x(t) dt. \tag{2.29}$$

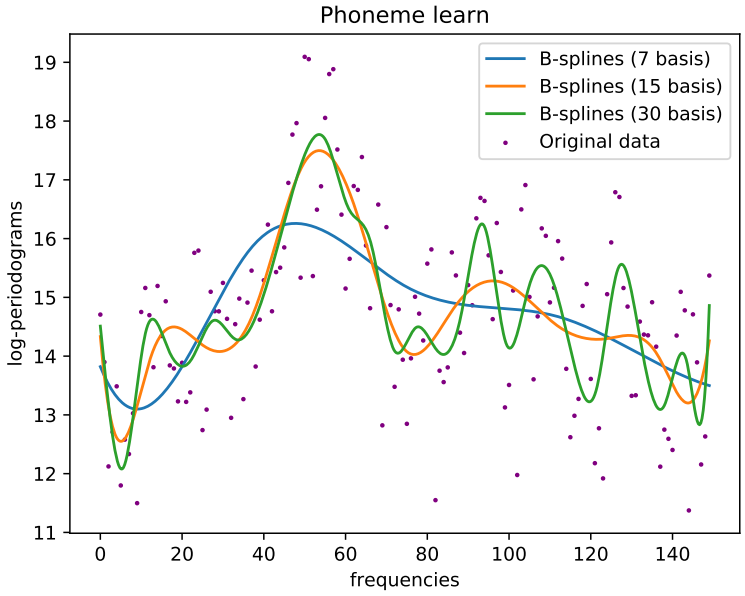


Figure 2.9: Basis representation of a sample of the phoneme dataset using B-splines with different number of basis functions.

The notation $D^m x(t)$ is used to indicate the m-th order derivative of the function $x(t)$. If a function has very high local variation, its curvature will have high values in an important part of its domain. In consequence, the value of the roughness $PEN_2(x)$ will be high.

Assume that we want to consider functions with a smooth first derivative. We can adapt the definition of roughness so that it considers the curvature of $Dx(t)$

$$PEN_3(x) = \int D^3 x(t) dt.$$

More general penalties can be defined as

$$PEN_L(x) = \int L(x(t)) dt \tag{2.30}$$

where L is any linear differential operator. In most cases, the penalties considered involve the m order derivative of the function $L(x) = D^m x$. [12]

Now we are ready to define the **penalized sum of squared errors**, a fitting criterion that combines roughness penalty (Equation (2.30)) and weighted SSE (Equation (2.27))

$$PENSSE_{L,W}(\boldsymbol{y}, \boldsymbol{c}) = \left(\boldsymbol{y} - \boldsymbol{c}'\Phi\right)' W \left(\boldsymbol{y} - \boldsymbol{c}'\Phi\right) + \lambda \int L(x(t)) dt \tag{2.31}$$

where λ is the **smoothness parameter** that controls the weight of the roughness penalty in the fitting criterion.

We can right the roughness penalty in as a matrix expression

$$\begin{aligned}
\int \left(Lx(t)\right)^2 dt &= \int \left(L\boldsymbol{c}'\phi(t)\right)^2 dt \\
&= \int \boldsymbol{c}' L\phi(t) L\phi'(t) \boldsymbol{c} \quad dt \\
&= \boldsymbol{c}' \left(\int L\phi(t) L\phi'(t) dt \right) \boldsymbol{c} \\
&= \boldsymbol{c}' R \boldsymbol{c}
\end{aligned}$$

where R is called **penalty matrix**

$$R = \int L(\phi(t)L(\phi(t))'dt. \tag{2.32}$$

We can then rewrite Equation (2.31) as a matrix equation

$$PENSSE_{L,W}(\boldsymbol{y}, \boldsymbol{c}) = \left(\boldsymbol{y} - \boldsymbol{c}'\Phi\right)' W \left(\boldsymbol{y} - \boldsymbol{c}'\Phi\right) + \lambda \boldsymbol{c}' R \boldsymbol{c} \tag{2.33}$$

The next step consist in differentiating, setting the derivative equal to 0 and solving $\boldsymbol{c}$ as done in the previous cases. The closed-form expression for the value of $\boldsymbol{c}$ that minimises Equation (2.31) is

$$\hat{\boldsymbol{c}} = \left(\Phi' W \Phi + \lambda R\right)^{-1} \Phi' W \boldsymbol{y}. \tag{2.34}$$

The result of applying roughness penalty to the fitting of a sample of the phoneme dataset in a B-splines system with 50 basis functions can be observedn in Figure 2.10. In this case the penalty term involves the second derivative ($L = D^2$). The value of the smoothness parameter used is $\lambda = 100$. Different values of the smoothness parameter in a logarithm scale should be considered to obtain the desired amount of smoothing [12].

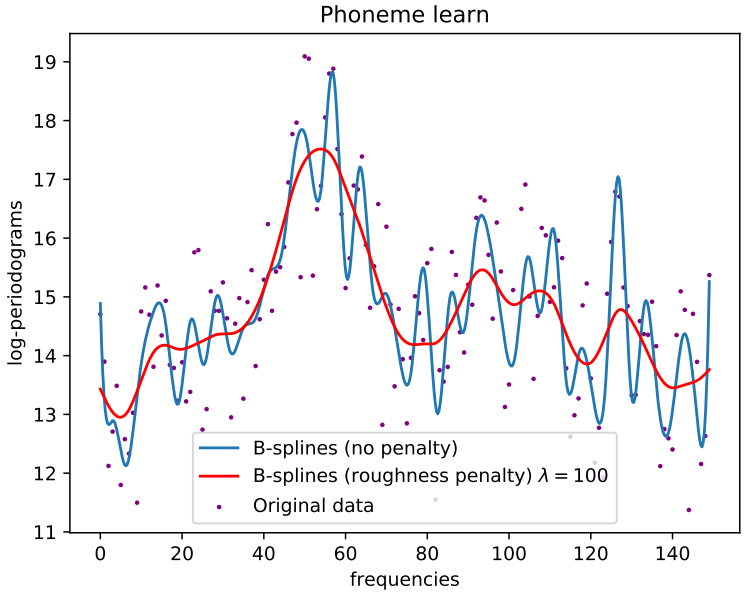


Figure 2.10: Fitting a sample of the phoneme dataset with roughness penalty.

2.3.3 Basic statistics

As we already said in Section 2.2.1 one of the first operations when analysing new data is to calculate its basic statistics, such as the mean or variance. As a reminder their formal definitions are given anew

Let $\{x_i(t)\}_{i=1,\dots,n}$ a set of functional data defined in the same functional space. The following statistics are defined:

Mean

$$\bar{x}(t) = \frac{1}{n} \sum_{i=1}^{n} x_i(t) \tag{2.35}$$

Variance

$$\sigma^2(t) = \frac{1}{n}\sum_{i=1}^{n}\left(x_i(t) - \bar{x}(t)\right)^2 \tag{2.36}$$

Covariance

$$\Sigma(s,t) = \frac{1}{n}\sum_{i=1}^{n}\left(x_i(s) - \bar{x}(s)\right)\left(x_i(t) - \bar{x}(t)\right) \tag{2.37}$$

Geometric mean

$$\left(\prod_{i=1}^{n} x_i(t)\right)^{\frac{1}{n}} \tag{2.38}$$

Given that we are representing functional data as a linear combination of basis functions $\phi_1(t), \phi_2(t), ..., \phi_K(t)$

$$x_i(t) = \sum_{k=1}^{K} c_{ik}\phi_k(t)$$

these statistics are computed as follows:

Mean

$$\bar{x}(t) = \frac{1}{n}\sum_{i=1}^{n}\sum_{k=1}^{K} c_{ik}\phi_k(t) = \sum_{k=1}^{K}\left(\frac{1}{n}\sum_{i=1}^{n} c_{ik}\right)\phi_k(t) \tag{2.39}$$

Covariance

$$\begin{aligned}
\Sigma(s,t) &= \frac{1}{n}\sum_{i=1}^{n}\left(\sum_{k=1}^{K} c_{ik}\phi_k(s) - \sum_{k=1}^{K}\bar{c}_k\phi_k(s)\right)\left(\sum_{k=1}^{K} c_{ik}\phi_k(t) - \sum_{k=1}^{K}\bar{c}_k\phi_k(t)\right) \\
&= \frac{1}{n}\sum_{i=1}^{n}\left(\sum_{k=1}^{K}\left(c_{ik} - \bar{c}_k\right)\phi_k(s)\right)\left(\sum_{k=1}^{K}\left(c_{ik} - \bar{c}_k\right)\phi_k(t)\right) \\
&= \frac{1}{n}\sum_{i=1}^{n}\sum_{k=1}^{K}\sum_{l=1}^{K}\left(c_{ik} - \bar{c}_k\right)\left(c_{il} - \bar{c}_l\right)\phi_k(s)\phi_l(t) \\
&= \sum_{k=1}^{K}\sum_{l=1}^{K}\left(\frac{1}{n}\sum_{i=1}^{n}\left(c_{ik} - \bar{c}_k\right)\left(c_{il} - \bar{c}_l\right)\right)\phi_k(s)\phi_l(t)
\end{aligned} \tag{2.40}$$

where

$$\bar{c}_k = \frac{1}{n}\sum_{m=1}^{n} c_{mk}$$

The variance is simply a specific case of the covariance: $\sigma^2(t) = \Sigma(t,t)$.

Geometric mean

$$\left(\prod_{i=1}^{n}\sum_{k=1}^{K} c_{ik}\phi_k(t)\right)^{\frac{1}{n}} = \exp\left(\frac{1}{n}\sum_{i=1}^{n}\log\left(\sum_{k=1}^{K} c_{ik}\phi_k(t)\right)\right. \tag{2.41}$$

The purpose of deriving this formulae of the statistics is to determine whether they can be represented in the same basis system as the original data. The mean can be represented as a linear combination of the basis functions. The covariance is not in the same function space of the data. If the data live in a function space $\mathcal{E}(D \longmapsto I)$, the covariance belongs to a function space $\mathcal{E}'(D \times D \longmapsto I)$. A basis system for this space is the cross product of the original basis

system times itself. The coefficients of the covariance in this direct product basis system are obtained through Equation (2.3.3). Notice that the vector of coefficients of the mean is the mean of the coefficients corresponding to each basis function and that the coefficients matrix of the covariance in the basis system $\phi \times \phi$ is actually the covariance matrix of the coefficients. As for the geometric mean, it cannot be represented by a linear expansion of additions or products of the original basis system.

2.3.4 Differentiation

A functional datum is represented by the following basis expansion

$$x(t) = \sum_{k=1}^{K} c_k \phi_k(t).$$

Differentiating the above formula we obtain

$$Dx(t) = \sum_{k=1}^{K} c_k D\phi_k(t). \tag{2.42}$$

As we can see, one of the advantages of the basis system representation is that the derivatives can be exactly computed if the functions in the basis system can be differentiated analytically, which is the case of the most used basis systems and the ones implemented in the Python fda package: monomials, Fourier and B-splines.

The closed-forms for these derivatives are:

Monomial

$$\begin{aligned} D\phi_0(t) &= Dt^0 = 0 \\ D\phi_k(t) &= Dt^k = (k-1)t^{k-1} = (k-1)\phi_{k-1}(t) \text{ for } k > 0 \end{aligned}$$

Fourier

$$\begin{aligned} D\phi_0(t) &= D\left(\frac{1}{\sqrt{T}}\right) = 0 \\ D\phi_{2n-1}(t) &= D\left(\sqrt{\frac{2}{T}}\sin\left(\frac{2\pi n}{T}t\right)\right) = \frac{2\pi n}{T}\sqrt{\frac{2}{T}}\cos\left(\frac{2\pi n}{T}t\right) = \frac{2\pi n}{T}\phi_{2n}(t) \\ D\phi_{2n}(t) &= D\left(\sqrt{\frac{2}{T}}\cos\left(\frac{2\pi n}{T}t\right)\right) = -\frac{2\pi n}{T}\sqrt{\frac{2}{T}}\sin\left(\frac{2\pi n}{T}t\right) = -\frac{2\pi n}{T}\phi_{2n-1}(t) \end{aligned}$$

BSplines

$$\begin{aligned} DB_{k,l,\tau}(t) &= D\left(\frac{t-t_k}{t_{k+l}-t_k}B_{k,l-1,\tau}(t) + \frac{t_{k+l+1}-t}{t_{k+l+1}-t_{k+1}}B_{k+1,l-1,\tau}(t)\right) \\ &= l\left(\frac{B_{k,l-1,\tau}(t)}{t_{k+l}-t_k} - \frac{B_{k+1,l-1,\tau}(t)}{t_{k+l+1}-t_{k+1}}\right) \end{aligned}$$

3

Development of the fda Python package

A general overview of Functional Data Analysis has been given in the previous chapter. Emphasis has been made in the description of the methods that have been implemented in the Python package fda. In this we describe the software development process. The analysis and design of the package and the tools used to ensure the quality of the software developed: unit testing, documentation, version control and continuous integration.

3.1 Analysis

The goal is to develop an open-source package to give support to Functional Data Analysis in Python.

The package is intended for data analysts, statisticians and scientist that either work on the field of FDA or may benefit from using it for data analysis and machine learning.

Given the goal of the project, the requirements are:

- The software developed has to be a Python package.
- The software follows Python coding and documenting standards defined in PEP 8 and PEP 257.
- The package needs to include as many FDA functionalities as possible given the alloted time for the project (360 hours).
- It has to be an open-source project.
- As the project is intended for a very general audience, the documentation has to be very thorough.
- The software has to be scalable and it has to allow an easy mechanism for contributing to it.
- To ensure the satisfaction of the previous point the project has to include and extensive test bench of unit test and continuous integration mechanisms.

- All algorithms used must be carefully designed for efficiency. The use of already programmed algorithms in the core mathematical Python libraries *numpy* [18] and *scipy* [19] is recommended when possible as they are well designed and programmed in lower level languages like C, Fortran or Cython which makes them very efficient.

3.2 Design

The package structure is built around two classes: *FDataGrid* and *FDataBasis*. They are the Python structures for discrete and basis representation of functional data as described in Sections 2.1.1 and 2.1.2. These two classes include methods for computing the basic statistics and methods to change from one representation to the other. The rest of the package is structured in different modules that are collections of the functionalities described in the previous chapter.

3.2.1 FDataGrid

It is the Python class corresponding to the discrete representation described in Section 2.1.1. It can represent functional data of any dimension. It consists in a tensor for storing the sampling data and a list of lists to store the sampling points. The cross product of each list of the later is a representation of the sampling space. As for the tensor of data, its first dimension represents the different samples, the following k are for each one of the dimension of the domain and the last is for the image and it has the same length as the dimension of the image.

For example a FDataGrid object representing a function $f : \mathbb{R}^k \longmapsto \mathbb{R}^l$ for which samples have been taken over a grid of shape $m_1 \times m_2 \times ... \times m_k$ sampling points for each dimension of the domain, has the following attributes:

- The sampling points structure, a length k list containing lists of length $m_1, m_2, ..., m_k$ whose cross product represent the sampling space.
- The data structure, a tensor of shape $n \times m_1 \times m_2 \times ... \times m_k \times l$ where n is the number of samples.

This class includes the *mean*, *var*, *cov* and *gmean* methods to compute the mean, variance, covariance and geometric mean as explained in Section 2.2.1. *derivative* returns the derivative of the given order as seen in Section 2.2.2. Methods *plot* and *scatter* for plotting the functional data like show in figure 2.1. It also includes the method *to_ basis* that given a *Basis* object (that will be later presented) returns a *DataBasis* with the basis representation of the *FDataGrid* object. All this methods work for FDataGrid object representing functions of the form $f : \mathbb{R} \longmapsto \mathbb{R}$ but not all of them work for higher dimensional functional data.

3.2.2 Basis

Before defining the *FDataBasis* class we need to give a representation to basis function systems. A *Basis* class is an abstract class providing a template for functional basis implementations.

It is designed only for functions that go from an interval of $\mathbb{R}$ to $\mathbb{R}$. The basic parameters to define any basis function system is the interval of $\mathbb{R}$ where the basis is defined and the number of basis functions although some implementation of *Basis* need more parameters.

It has a method *evaluate* that given a list of points, evaluates each basis function of the basis system at the given points. It also evaluates the derivatives of the basis functions if the

argument *derivative* is given. For plotting there is the method *plot* which also admits a *derivative* argument. It also defines the method *penalisation* that given a linear differential operator return the roughness penalty matrix R as defined by Equation (2.30). The computing of R is exact when the linear differential operator is a derivative if the basis functions admit analytical differentiation (which is the case of all the basis systems implemented). If the differential operator is more complex numerical approximations are used.

Three implementations of this abstract class exist: **Monomial**, **Fourier** and **BSpline** corresponding to the basis explained in the theoretical framework in Section 2.1.2. *Fourier* only admits and odd number of basis functions and it needs an extra parameter *period* to be defined. *BSpline* also takes an extra parameter indicating where to place the knots.

3.2.3 FDataBasis

It the Python class corresponding to the basis representation of functional data as explained in Section 2.1.2. As the *Basis* class, it is only designed for functions $f : [a, b] \subset \mathbb{R} \longmapsto \mathbb{R}$.

Its composed of two elements, a *Basis* object with K basis functions and a matrix of shape $n \times K$ where n is the number of samples. Each row of the matrix consists in the coefficients that multiplied by the basis functions represent each of the samples.

As the *FDataGrid* class, it has methods for basic statistics *mean*, *cov* and *gmean*. As seen in Section 2.3.3 the mean can be represented as a FDataBasis object with the same basis system. The covariance would need bivariate FDataBasis, as it is not yet implemented, the exact representation of the covariance rests as future work and a numerical approximation is returned by transforming the *FDataBasis* object in to its discrete representation. The same solution is applied for the geometric mean. As each sample in a *FDataBasis* is a function, they can be evaluated. For this purpose there is the method *evaluate* that given a list of points it evaluates each of the sample at them. The class also implements the special method *__call__* that makes every instance of the class a function that can be called. This method is just a proxy of *evaluate*. To change to a discrete representation there is the method *to_grid* that evaluates the object at a given sampling points and builds a *FDataBasis* object.

The most complex method of *FDataBasis* is its **class method** *from_data*. It implements the fitting and roughness penalty smoothing detailed in Sections 2.3.1 and 2.3.2. Given a matrix with samples of discrete functional data and a list of the sampling points of these data it fits them to a functional basis system represented by a given *Basis* function returning a *FDataBasis* object. If additionally a linear differential operator and a smootheness parameter are given roughness penalty smoothing is applied to the fit. The method *to_basis* of *FDataGrid* mentioned in Section 3.2.1 is just a wrapper of this method.

The last method requires solving a least squares problem. Two different algorithms are implemented, one using Cholesky which is faster and another using QR factorisation which is more stable. The description of these two algorithms is in appendix A.

3.2.4 kernels, kernel_smoothers and validation modules

These three modules together give support to the kernel smoothing functionality described in Section 2.2.3.

The *kernel* and *kernel_smoothers* modules define all the kernel functions and kernel smoothers that the package provides. An exhaustive list of them can be found in appendix A.

The *validation* module implement the different cross-validation scores used for choosing the best smoothness parameter. It also implements the function *minimise* that given a set

of smoothness parameters, a cross-validation criterion, a kernel smoother with a kernel and a dataset returns the smoothness parameter and its corresponding smoothing matrix and smoothed data set that minimises the chosen cross validation score.

3.2.5 math module

The *math* module includes the functions *mean*, *var*, *cov* and *gmean* that are just wrappers of the methods with the same name in both functional data classes. It also includes a series of transformations like *sqrt*, *log2*, *log*, *log10*, *exp* that can be applied to a *FDataGrid* object to obtain its square root, logarithm or exponential. It also includes the functions *inner_product*, *norm_lp* and *metric* to calculate the $\mathcal{L}_p$ norm for $1 \leq p \leq \infty$, the distance and inner product between *FDataGrid* objects. The FPCA algorithm described in Section 2.2.5 is implemented in the *fpca* function.

For a more detailed description of all these functions, methods and classes the whole fda package documentation is included in appendix C and it is also available on-line in the *read the docs* website of the project: `fda.readthedocs.io`.

3.3 Coding, documenting and testing

When it comes to coding, specially if more than one person is going to be involved, it is important to think about the readability of the code and to program following style standards so that the style of the code and documentation remains uniform all throughout the package. For this reason we have made sure that we have followed the most important style guides for Python: **PEP 8 – Style Guide for Python Code** [20] and **PEP 257 – Docstring Conventions** [21]. To ensure the correctness of the code according to these two standards the tool **prospector** [22] was used. Among other functionalities this tool looks for coding style convections violations in the code.

PEP 8 is the *de facto* style standard for Python code whereas PEP 257 is the standard for Python **docstrings**. A *docstring* is according to PEP 257 "*a string literal that occurs as the first statement in a module, function, class, or method definition. Such a docstring becomes the* `__doc__` *special attribute of that object.*" [21]. *Docstrings* are the core of any Python package documentation, they are used to describe modules, functions, classes and methods. Together with (typically) **reStructuredText** files used for building indexes and giving a more general description of the package; they conform the whole documentation source of a Python package.

Apart from PEP 257 other *docstring* more structured standards exist that were built over the former one. The most popular are the *NumPy* and *Google* ones. The former is used in the most popular and used Python package like *NumPy*, *Scipy* and *pandas* and has a syntax based on *reStructuredText*. The later is more recent and has less spread but it has a more *'pythonic'* syntax based on indentations. This is the reason why we chose the later one. We found it easier to read and matches better with the code it documents.

One of the advantages of following all these documentation standards is that tools exist for compiling this source documentation into more readable formats such as *PDF* or *HTML*. The most used tool and our choice for compiling documentation in Python is **Sphinx**. It provides a very complete framework to perform this task and among other things it is able to capture and render LaTeX-like syntax from the *docstrings* and other files which is very useful for mathematical projects like this one. The documentation found in appendix C was built using *Sphinx*.

Another fundamental part of producing high quality software is unit testing. It consists in elaborating test benches for each individual piece of the software, in this case, every method and

function. There exists several tools for unit testing in Python. Our choice was **py.test** because it is very easy to use and configure and provides everything we needed. More sophisticated tools exist such as **tox** that allow for instance running test benches with different Python interpreters. This tool was discarded because it is harder to configure and the extra functionalities that come with it were not needed. Although, if support for different versions of the Python language will be needed, this tool may come handy.

We have talked about documenting and unit testing as two separated tasks in the process of building the software. Nevertheless the Python standard libraries include the **doctest** module. This module searches for pieces of what looks like a Python interactive console inside the *docstrings* and runs them and checks that they produce the same output as specified by the *docstring*. This is commonly used to give examples to a function, class or method and build unit test for that same piece of software at the same time.

In the box below we can see the *docstring* of the *Monomial* class in the *basis* module. Apart from the the class general description, it has an *Attributes* section and an *Examples* one. The later includes some lines starting with '$>>>$'. These lines together with the ones right below them form the *doctests*. Each of these lines is executed and compared with the line below that represents its expected output. If it matches the test passes, if not the test is considered to fail.

```
class Monomial(Basis):
    """Monomial basis.

    Basis formed by powers of the argument :math:`t`:

    .. math::
        1, t, t^2, t^3...

    Attributes:
        domain_range (tuple): a tuple of length 2 containing the initial and
            end values of the interval over which the basis can be evaluated.
        nbasis (int): number of functions in the basis.

    Examples:
        Defines a monomial base over the interval :math:`[0, 5]` consisting
        on the first 3 powers of :math:`t`: :math:`1, t, t^2`.

        >>> bs_mon = Monomial((0,5), nbasis=3)

        And evaluates all the functions in the basis in a list of descrete
        values.

        >>> bs_mon.evaluate([0, 1, 2])
        array([[1., 1., 1.],
               [0., 1., 2.],
               [0., 1., 4.]])

        And also evaluates its derivatives

        >>> bs_mon.evaluate([0, 1, 2], derivative=1)
        array([[0., 0., 0.],
               [1., 1., 1.],
               [0., 2., 4.]])
        >>> bs_mon.evaluate([0, 1, 2], derivative=2)
        array([[0., 0., 0.],
               [0., 0., 0.],
               [2., 2., 2.]])

    """
```

In figure 3.1 we can see how *Sphinx* treats the *docstring* above, rendering the mathematical

input, giving a bold font to section headers and putting *doctests* in boxes emulating consoles.

Examples

Defines a monomial base over the interval $[0,5]$ consisting on the first 3 powers of t: $1, t, t^2$.

```
>>> bs_mon = Monomial((0,5), nbasis=3)
```

And evaluates all the functions in the basis in a list of descrete values.

```
>>> bs_mon.evaluate([0, 1, 2])
array([[1., 1., 1.],
       [0., 1., 2.],
       [0., 1., 4.]])
```

And also evaluates its derivatives

```
>>> bs_mon.evaluate([0, 1, 2], derivative=1)
array([[0., 0., 0.],
       [1., 1., 1.],
       [0., 2., 4.]])
>>> bs_mon.evaluate([0, 1, 2], derivative=2)
array([[0., 0., 0.],
       [0., 0., 0.],
       [2., 2., 2.]])
```

Figure 3.1: HTML version of the *Examples* section in *fda.basis.Monomial* class *docstring*.

3.4 Version control, repositories and continuous integration

One of the requirements of the project was that it had to be open-source. Version control and continuous integration are really important in any project that aims to survive their first release but it becomes crucial in an open-source project because the potential number of contributors is big and the communication among then is very little. Protocols for contributing to the project, reviewing and accepting contributions and generating new releases have to be established. Fortunately there are tools available for free for open-source projects that are very useful for managing the flow of contributions.

As a version control system we are using **git** that apart from its simplicity it comes with the great advantage that hosting services for version control using git such as **GitHub** or **Bitbucket** exist. The open-source community choice is usually *GitHub* because it is free and it comes with all its functionalities for public repositories. A public repository can be read and cloned by anyone but writing is controlled by the owners of the repository. It is the kind you want for carrying out an open-source project. Another advantage of using *GitHub* is the great amount of tools that integrate with it and that are, of course, free for open-source projects. The URL of the *GitHub* page of the project is `https://github.com/mcarbajo/fda`.

Git is a very flexible tool, it allows to easily create versions or **commits** as they are known in the jargon and it allows to easily manage parallel lines of work called **branches**. This is a very simplistic description of what *git* does but it is enough for the purpose of this document.

Several workflows have been designed to help developers carry out version control with git in a structured way. One of the most popular and the one we have used in this project is **gitflow**. It structures the project in two main brances: *master* that will contain the different releases of the software and *develop* that contains the versions generated during the development. In addition

feature branches are created for developing new functionality and then they are merged to the *develop* branch. Before taking the versions generated in *develop* to *master* a *release* branch is created to carefully test the software before making the final release. If bugs are detected in *master*, *hotfix* branches are created to solve this issues and then they are merged back to *master*. This written explanation can be a bit confusing and the schema in figure 3.2 may help the reader understand the workflow [23].

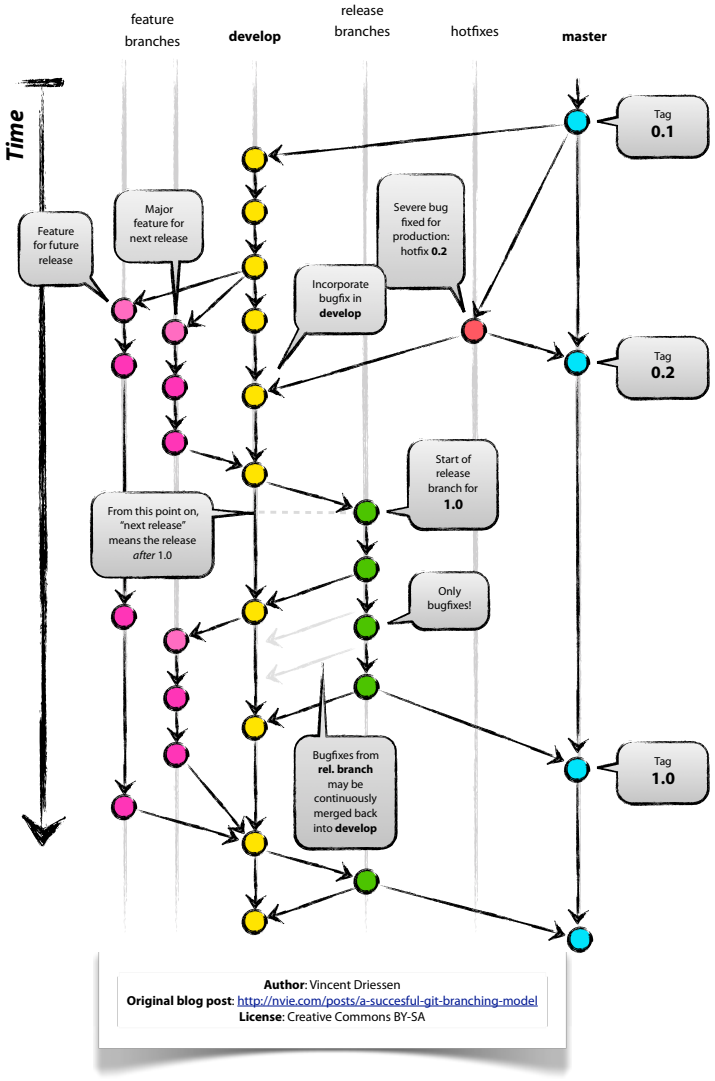


Figure 3.2: Gitflow schema.

In Section 3.3 we talked about how to ensure the quality of the software with unit tests and tools to made them such as *py.test*. The procedure of running all unit test and checking the coverage of them it is not a just one time task. Every time a new version is created all test should be run to ensure that it still meets the test defined previously. The practice of creating new versions and checking that they meet the same criteria defined for previous ones is called **continuous integration (CI)**. There are numerous tools for the automation of this task but **Travis CI** is a very usual choice for open-source projects on *GitHub*. *Travis CI* can either periodically clone the repository and run all the tests or do it every time a push is done to

GitHub. It then notifies the user if any errors are detected. It is very useful if the project gives support to more than one version of Python because it can be configured to run the test with different versions of the interpreter. Another advantage and one of the reasons it is so popular is that it is free for public *GitHub* repositories.

In addition to unit testing, the compilation of the documentation of the project can be also automatised thanks to the tool **Read the Docs**. This tool provides automated generation of documentation as well as free hosting for it. As *Travis CI* it also links to the *GitHub* repository and detects every time a new commit is available. Then it uses sphinx to compile it and then hosts the generated documentation in its website. The URL to the fda package documentation is `http://fda.readthedocs.io/en/latest/`.

Some development tasks, which are still done locally and carried out manually, should be performed at a global level in an automatic manner. In particular, checking for style conventions violations can be automatised with the help of the tool **landscape**, which similarly to the tools **Read the Docs** and *Travis CI*, links to the *GitHub* repository and performs checks every time a commit is done.

4

Conclusions and future work

The final goal of this project is the development of a comprehensive Python package for Functional Data Analysis. The contributions of this undergraduate thesis is to initiate the project and implement some basic functionality. In this section will summarily describe some essential features of the project and discuss the following steps in the construction of the fda package.

When starting a project from scratch is very important not to hurry and start writing code. It is essential to first analyse and design carefully what you are trying to build. For instance, one of the first decisions we had to made was how to represent functional data. Before any line of code of the fda Python package was written we had already gone through other FDA libraries for fda and determined the pros and cons of each of the representations.

The project scope was not fully delimited at the beginning. One of the major requirements was that project be open-source. These two facts meant that the developing process needed to be made iteratively, through gradual extensions in the functionality of the package. Nowadays many software development projects deal with similar issues. Many tools and work flows are available to address the difficulties and reduce the conflicts that may occur. In our case the use of *git* and *GitHub* for managing version control and *TravisCI* to do continuous integration have been fundamental for the success of the project.

A first release of the fde Python package will be made after some key functionalities are added in the most used repositories for Python packages: **PyPi** and **Anaconda Cloud**.

The next few steps before the release are related to preprocessing the data and exploratory analysis. This includes the completion of smoothing functionality for basis representation, allowing smoothing under constraints so that the resulting function is strictly positive or monotone. Also allowing the basis representation to be 2-dimensional in its domain to be able to represent the covariances. Another key part of treating with functional data is dealing with variation in phase and amplitude. A series of techniques usually referred to as **registration** deal with this issue. [12]

Actually FDA goes beyond exploratory analysis it includes machine learning techniques for regression and classification. Learning by automatic induction from functional data is a booming research area in which software tools are still lacking. Our long term goal is to implement these novel techniques so that the Pyhton fda package evolves together with the field of Functional Data Analysis.

Acronyms and glossary

4.1 Acronyms and abbreviations

- **CI**: Continuous integration
- **CV**: Cross validation
- **FDA**: Functional data analysis
- **FPCA**: Functional Principal Component Analysis
- **GCV**: General cross validation
- **KNN**: K-nearest neighbours
- **PCA**: Principal Component Analysis
- **PEN**: Penalisation
- **PENSSE**: Penalised sum of square errors
- **SSE**: Sum of square errors
- **SVD**: Singular value decomposition

4.2 Glossary

- **Class method**: In Python a class method (marked with the decorator @classmethod) is a method whose first argument *cls* is a reference to the class of the method. It is often used to create alternative constructors apart from $\_\_init\_\_$.
- **Docstring**: In Python a docstring is a string literal that occurs as the first statement in a module, function, class, or method definition. It is used to document each singles piece of a Python software.

Bibliography

[1] U. Grenander. Stochastic processes and statistical inference. *Arkiv för Matematik*, 1:195–277, 1950.

[2] C.R. Rao. Some statistical methods for comparison of growth curves. *Biometrics*, 14:1–17, 1958.

[3] J.O. Ramsay. When the data are functions. *Psychometrika*, 47:379–396, 1982.

[4] J. Goldsmith, F. Scheipl, L. Huang, J. Wrobel, J. Gellar, J. Harezlak, M.W. McLean, B. Swihart, L. Xiao, C. Crainiceanu, P.T. Reiss, Y. Chen, S. Greven, L. Huo, M.G. Kundu, S.Y. Park, D.L. Miller, and A. Staicu. refund: Regression with functional data. `https://CRAN.R-project.org/package=refund`, August 2016.

[5] M. Febrero-Bande and M. Oviedo de la Fuente. Statistical computing in functional data analysis: The R package fda.usc. *Journal of Statistical Software*, 51(4):1–28, 2012.

[6] F. Yao, B. Liu, H. Müller, J. Wang, K. Chen, A. Gottlieb, S. Wu, A. Peterson, H. Ji, X. Dai, C. Chen, A. Farris, W. Tao, X. Zhang, J. He, C. Xu, R. Tang, W. Yang, J. Chiou, and J. Dubin. Pace: Principal analysis by conditional expectation. `http://www.stat.ucdavis.edu/PACE/`, June 2015.

[7] J.O. Ramsay and B. W. Silverman. Functional data analysis - software. `http://www.psych.mcgill.ca/misc/fda/software.html`, August 2017.

[8] S. Cass. The 2017 top programming languages. `https://spectrum.ieee.org/computing/software/the-2017-top-programming-languages`, July 2017.

[9] J.O. Ramsay, H. Wickham, S. Graves, and G. Hooker. *fda: Functional Data Analysis*, 2017. R package version 2.4.7.

[10] M. Febrero-Bande and M. Oviedo de la Fuente. Statistical computing in functional data analysis: The R package fda.usc. *Journal of Statistical Software*, 51(4):1–28, 2012.

[11] H.H. Thodberg. Tecator. `http://lib.stat.cmu.edu/datasets/tecator`.

[12] J.O. Ramsay and B.W. Silverman. *Functional Data Analysis*. Springer-Verlag New York, 2nd edition, 2005.

[13] I. J. Schoenberg. Contributions to the problem of approximation of equidistant data by analytic functions: Part a. on the problem of smoothing or graduation. a first class of analytic approximation formulae. *Quarterly of Applied Mathematics*, 4:45–99, 1946.

[14] C. de Boor. *A Practical Guide to Splines*. Springer-Verlag New York, 1st edition, 1978.

[15] Wasserman L. *All of Nonparametric Statistics*. Springer-Verlag New York, 1st edition, 2006.

[16] Phoneme. `http://www.math.univ-toulouse.fr/staph/npfda/npfda-datasets.html`.

[17] D.J. Levitin, N. Regina, V. Bradley, and J.O Ramsay. Introduction to functional data analysis. 48:135–155, 08 2007.

[18] T. Oliphant. *A Guide to NumPy*. Trelgol Publishing, 2006.

[19] E. Jones, T. Oliphant, and R. Peterson. SciPy: Open source scientific tools for Python, 2001.

[20] G. van Rossum, B. Warsaw, and N. Coghla. Pep 8 – style guide for python code, 2013.

[21] D. Goodger and G. van Rossum. Pep 257 – docstring conventions, 2001.

[22] C. Crowder, J. Simeone, J. Quast, S. Spilsbury, and J.A. Perdiguero. prospector. `https://github.com/PyCQA/prospector`, 2013.

[23] V. Driessenl. A successful git branching model, 2010.

A

Equations and algorithms

A.1 Kernels

In this section we list the kernel functions implemented and their closed-forms.

Normal

$$K(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}} \tag{A.1}$$

Epanechnikov

$$K(x) = \begin{cases} \frac{3}{4}(1-x^2) & \text{if } |x| \leq 1 \\ 0 & \text{elsewhere} \end{cases} \tag{A.2}$$

Cosine

$$K(x) = \begin{cases} \frac{\pi}{4} cos\left(\frac{\pi x}{2}\right) & \text{if } |x| \leq 1 \\ 0 & \text{elsewhere} \end{cases} \tag{A.3}$$

Tri-weight kernel

$$K(x) = \begin{cases} \frac{35}{32}\left(1-u^2\right)^3 & \text{if } |x| \leq 1 \\ 0 & \text{elsewhere} \end{cases} \tag{A.4}$$

Quartic kernel

$$K(x) = \begin{cases} \frac{15}{16}\left(1-u^2\right)^2 & \text{if } |x| \leq 1 \\ 0 & \text{elsewhere} \end{cases} \tag{A.5}$$

Uniform kernel

$$K(x) = \begin{cases} \frac{1}{2} & \text{if } |x| \leq 1 \\ 0 & \text{elsewhere} \end{cases} \tag{A.6}$$

A.2 Kernel smoothers

As mentioned in Section 2.2.4 all smoothers included in the package are linear, meaning that for each t, there exists a vector $l(t) = (l_1(t), l_2(t), ..., l_m(t))^T$ such that

$$\hat{x}(t) = \sum_{i=1}^{m} l_i(x) y_i \tag{A.7}$$

The definition of $l(t)$ depends on the kernel smoother choice. As seen is Section 2.2.3 the smoothing problem ca be rewritten as as matrix product $\hat{Y} = \hat{H}Y$ where $\hat{H}_{ij} = l_j(t_i)$ and Y is the observations matrix.

The following kernel smoothers are the ones implemented in the Python package.

A.2.1 Nadaraya-Watson

Given a kernel the Nadaraya-Watson kernel smoother is defined as:

$$\hat{H}_{i,j} = \frac{K\left(\frac{t_i - t_j}{h}\right)}{\sum_{k=1}^{n} K\left(\frac{t_1 - t_k}{h}\right)} \tag{A.8}$$

where $K(\cdot)$ is the kernel function and h the kernel window width.

A.2.2 Local Linear Regression

Given a kernel the local linear regression kernel smoother is defined as:

$$\begin{aligned} \hat{H}_{i,j} &= \frac{b_i(t_j)}{\sum_{k=1}^{n} b_k(t_j)} \\ b_i(xt) &= K\left(\frac{t_i - t}{h}\right) S_{n,2}(t) - (t_i - t) S_{n,1}(t) \\ S_{n,k}(t) &= \sum_{i=1}^{n} K\left(\frac{t_i - t}{h}\right) (t_i - t)^k \end{aligned} \tag{A.9}$$

where $K(\cdot)$ is the kernel function and h the kernel window width.

A.3 CV methods

Although the most used score for smoothing parameter selection is the GCV score. Others are implemented. Here we provide the complete list.

A.3.1 Cross-validation

The CV closed-form is:

$$CV(\nu) = \frac{1}{n}\sum_j \left(\frac{y_j - \hat{y}_j^\nu}{1 - \hat{H}_{jj}^\nu}\right)^2 \tag{A.10}$$

where the hat matrix $\hat{H}^\nu$ and the estimation depend on the parameter ν.

A.3.2 General cross-validation

The general cross validation score (GCV) is defined by

$$GCV(\nu) = \frac{1}{n}\sum_{j=1}^{m} \left(\frac{y_j - \hat{y}_j^\nu}{1 - tr(\hat{H}^\nu)/m}\right)^2 \tag{A.11}$$

where the hat matrix $\hat{H}^\nu$ and the estimation depend on the parameter ν.

More general cross validation scores can be defined. Given a penalisation function $\Xi(\nu, n)$ we have

$$GCV(\nu) = \Xi(\nu, n)\frac{1}{n}\sum_j \left(y_j - \hat{y}_j^\nu\right)^2 \tag{A.12}$$

Akaike's information criterion

$$\Xi(\nu, n) = \exp\left(2\frac{tr(\hat{H}^\nu)}{n}\right) \tag{A.13}$$

Finite prediction error

$$\Xi(\nu, n) = \frac{1 + \frac{tr(\hat{H}^\nu)}{n}}{1 - \frac{tr(\hat{H}^\nu)}{n}} \tag{A.14}$$

Shibata's model selector

$$\Xi(\nu, n) = 1 + 2\frac{tr(\hat{H}^\nu)}{n} \tag{A.15}$$

Rice's bandwidth selector for cross validation.

$$\Xi(\nu, n) = \left(1 - 2\frac{tr(\hat{H}^\nu)}{n}\right)^{-1} \tag{A.16}$$

A.4 Smoothing with roughness penalty

In Section 2.3.2 we were left with the problem of solving Equation (2.34):

$$\left(\Phi' W \Phi + \lambda R\right)\boldsymbol{c} = \Phi' W \boldsymbol{y}$$

where $\boldsymbol{c}$ is the unknown. For simplicity in notation we define $A = \Phi' W \Phi + \lambda R$ and $\boldsymbol{b} = \Phi' W \boldsymbol{y}$ so we are left with the equation

$$A\boldsymbol{c} = \boldsymbol{b}.$$

In general, this equation has no exact solution but we want to obtain the closest thing to a solution that is possible. We are treating with a linear least squares fitting problem. For these purpose we set the normal equation

$$A'A\boldsymbol{c} = A'\boldsymbol{b}.$$

By solving it we obtain the solution for the least squares fitting problem.

There are several algorithms for solving this later equation. In the fda package two were implemented.

A.4.1 Cholesky decomposition

Cholesky factorisation is an algorithm for decomposing a Hermitian symmetric positive definite matrix into a product of an upper (or lower) matrix and its conjugate transpose

$$M = U^*U.$$

Note that A is a real full rank matrix so $A'A$ meets all the criteria to be decomposed by the algorithm

$$A'A = U'U.$$

Now we can proceed to the algorithm:

1. Calculate $A'A$ and $A'\boldsymbol{b}$ to obtain the normal equation.
2. Decompose $A'A$ into $U'U$ using cholesky factorisation.
3. Solve $\boldsymbol{x}$ in the lower triangular system $U'\boldsymbol{x} = A'\boldsymbol{b}$.
4. Finally, solve $\boldsymbol{c}$ in the upper triangular system $U\boldsymbol{c} = \boldsymbol{x}$.

Actually the algorithm was not implemented in the fda package and the *scipy* package implementation was used. The order of the algorithm is $\mathcal{O}(mn^2 + \frac{1}{3}n^3)$ where A is a $m \times n$ matrix.

A.4.2 QR factorisation

Reduced QR factorisation is an algorithm for decomposing a $m \times n$ matrix A into a product of an $m \times n$ orthogonal matrix Q and an $n \times n$ upper triangular matrix R

$$A = QR.$$

The algorithm for solving the least squares problem is:

1. Compute $A'A$.
2. Compute the reduce QR factorisation of A, $A = QR$.
3. Compute $Q'\boldsymbol{b}$.
4. Solve $\boldsymbol{c}$ in the upper triangular system $R\boldsymbol{c} = Q'\boldsymbol{b}$.

As it was done for the Cholesky decomposition algorithm the *scipy* implementation was used. The order of the algorithm is $\mathcal{O}(2mn^2 - \frac{2}{3}n^3)$ where A is a $m \times n$ matrix.

Cholesky decomposition is faster and it is the recommended choice. QR factorisation should be used when the rounding error in the Cholesky decomposition is too big. For this reason when using this later algorithm the Equation (2.34) is transformed to another one that is better from the standpoint of the rounding error.

The roughness penalty matrix R is symmetric and positive definite so it can be decomposed into a product of an upper matrix and its transposed.

$$R = U'U.$$

Now we can define

$$\tilde{\boldsymbol{y}} = \begin{bmatrix} \boldsymbol{y} \\ \mathbf{0} \end{bmatrix}$$

where $\mathbf{0}$ has the same length as the coefficients vector $\boldsymbol{c}$.

Let $\tilde{\Phi}$ be

$$\tilde{\Phi} = \begin{bmatrix} \Phi \\ \lambda U \end{bmatrix}$$

and $\tilde{W}$ be

$$\tilde{W} = \begin{bmatrix} \Phi & 0 \\ 0 & I \end{bmatrix}$$

where I is the identity matrix of size of the same length as the coefficients vector.

The resulting equation that is resolved using QR factorisation looks like

$$\hat{\boldsymbol{c}} = \left(\tilde{\Phi}'\tilde{W}\tilde{\Phi}\right)^{-1}\tilde{\Phi}'\tilde{W}\tilde{\boldsymbol{y}}. \tag{A.17}$$

B

Scripts

This appendix provides the Python scripts that using the fda package generate the results and graphics used throughout the whole document.

B.1 Discrete representation of functional data

This script generates the graphs of the functional data samples used to illustrate Section 2.1.1.

```
import os
import numpy as np
import matplotlib.pylab as plt

from fda import FDataGrid


if __name__ == '__main__':

    # Tecator data is structured in 3 differents csv files. One containing
    # the data at the sample points. Other containing the sample points. And
    #  a last one containing information about the percentage of fat,
    # water and protein in each sample.
    _dir = os.path.dirname(__file__)

    # Loads all 3 csv files.
    data = np.genfromtxt(os.path.join(_dir, '../data/tecator_data.csv'),
                         delimiter=',',
                         skip_header=1)

    sample_points = np.genfromtxt(
        os.path.join(_dir, '../data/tecator_sample_points.csv'),
        delimiter=',',
        skip_header=1)

    y = np.genfromtxt(os.path.join(_dir, '../data/tecator_y.csv'),
                      delimiter=',',
                      skip_header=1)

    # Builds a FDataGrid object using the loaded information.
    fd = FDataGrid(data, sample_points,
```

```
                    dataset_label='Spectrometric curves',
                    axes_labels=['Wavelength (mm)', 'Absorbances'])

    fd = fd[:5]
    # Plots the first 5 samples in a scatter plot.
    plt.figure()
    fd.scatter(s=0.5)

    # Plots the first 5 samples in a line plot.
    plt.figure()
    fd.plot()
    plt.show()
```

B.2 Basis representation

This script generates the graphs of the function basis systems in Section 2.1.2.

```
from fda.basis import Fourier, BSpline

import numpy as np
import matplotlib.pyplot as plt
from matplotlib import rc
rc('text', usetex=True)

if __name__ == '__main__':

    plt.figure()
    Fourier(nbasis=5).plot()
    plt.title('Fourier series system')
    plt.legend([r'$\phi_0 = 1$',
                r'$\phi_1 = \sqrt{2} sin(2 \pi t)$',
                r'$\phi_2 = \sqrt{2} cos(2 \pi t)$',
                r'$\phi_3 = \sqrt{2} sin(4 \pi t)$',
                r'$\phi_4 = \sqrt{2} cos(4 \pi t)$'])

    plt.figure()
    Fourier(nbasis=5).plot(derivative=1)
    plt.title('Fourier series system. First derivative')
    plt.legend([r"$\phi_0' = 0$",
                r"$\phi_1' = 2\pi\sqrt{2} cos(2 \pi t)$",
                r"$\phi_2' = -2\pi\sqrt{2} sin(2 \pi t)$",
                r"$\phi_3' = 4\pi\sqrt{2} cos(4 \pi t)$",
                r"$\phi_4' = -4\pi\sqrt{2} sin(4 \pi t)$"])

    plt.figure()
    bs = BSpline(knots=np.linspace(0, 1, 7), order=4)
    bs.plot()
    for knot in bs.knots:
        plt.axvline(knot, linestyle='--', linewidth=1)
    plt.title('B-spline system')
    plt.show()
```

B.3 Kernel smoothing

This script generates the graphs of the examples of kernel smoothing in Section 2.2.3.

```
""" Script to reproduce the example in the Kernel Smoothing section of the
end of degree project.
```

```
Uses different kernel smoothing methods over the phoneme data set and shows
how cross validations scores vary over a range of different parameters used in
the smoothing methods.

"""

import os
import numpy as np
import matplotlib.pylab as plt

import fda
import fda.validation as val
import fda.kernel_smoothers as ks

from fda import kernels


if __name__ == '__main__':

    # Kernels examples
    t = np.linspace(-2, 2, 501)
    plt.figure()
    plt.plot(t, kernels.normal(t))
    plt.plot(t, kernels.epanechnikov(t))
    plt.plot(t, kernels.uniform(t))

    plt.title('Kernels')
    plt.legend(['Normal', 'Epanechnikov', 'Uniform'])

    # Loads the phoneme data set from a csv file and build a FDataGrid object.
    _dir = os.path.dirname(__file__)

    data = np.genfromtxt(os.path.join(_dir, '../data/phoneme_data.csv'),
                         delimiter=',',
                         skip_header=1)
    fd = fda.FDataGrid(data, list(range(data.shape[1])),
                       dataset_label='Phoneme learn',
                       axes_labels=['frequencies', 'log-periodograms'])

    # Plots the first five samples of the data set.
    plt.figure(1)
    fd[0:5].plot()

    # Calculates general cross validation scores for different values of the
    #  parameters given to the different smoothing methods.

    # Local linear regression kernel smoothing.
    llr = val.minimise(fd, list(np.linspace(2, 25, 24)),
                       smoothing_method=ks.local_linear_regression)
    # Nadaraya-Watson kernel smoothing.
    nw = fda.validation.minimise(fd, list(np.linspace(2, 25, 24)),
                                 smoothing_method=ks.nw)
    # K-nearest neighbours kernel smoothing.
    knn = fda.validation.minimise(fd, list(np.linspace(2, 25, 24)),
                                  smoothing_method=ks.knn)

    # Plots the different scores for the different choices of parameters for
    #  the 3 methods.
    plt.figure(2)
    plt.plot(np.linspace(2, 25, 24), knn['scores'])
    plt.plot(np.linspace(2, 25, 24), llr['scores'])
    plt.plot(np.linspace(2, 25, 24), nw['scores'])
```

```
ax = plt.gca()
ax.set_xlabel('Smoothing method parameter')
ax.set_ylabel('GCV score')
ax.set_title('Scores through GCV for different smoothing methods')
ax.legend(['k-nearest neighbours', 'local linear regression',
           'Nadaraya-Watson'],
          title='Smoothing method')

# Plots the smoothed curves corresponding to the 11th element of the data
#  set (this is a random choice) for the three different smoothing methods.
plt.figure(3)
fd[10].plot()
knn['fdatagrid'][10].plot()
llr['fdatagrid'][10].plot()
nw['fdatagrid'][10].plot()
ax = plt.gca()
ax.legend(['original data', 'k-nearest neighbours',
           'local linear regression',
           'Nadaraya-Watson'],
          title='Smoothing method')

# Plots the same 5 samples from the beginning using the Nadaraya-Watson
# kernel smoother with the best choice of parameter.
plt.figure(4)
nw['fdatagrid'][0:5].plot()

# Undersmoothing and oversmoothing
fd_us = fda.FDataGrid(ks.nw(fd.sample_points[0], h=2)
                      @ np.squeeze(fd.data_matrix[10], axis=-1),
                      fd.sample_points, fd.sample_range, fd.dataset_label,
                      fd.axes_labels)
fd_os = fda.FDataGrid(ks.nw(fd.sample_points[0], h=15)
                      @ np.squeeze(fd.data_matrix[10], axis=-1),
                      fd.sample_points, fd.sample_range, fd.dataset_label,
                      fd.axes_labels)

# Not smoothed
plt.figure(5)
fd[10].plot()

# Smoothed
plt.figure(6)
fd[10].scatter(s=0.5)
nw['fdatagrid'][10].plot(c='g')

# Under-smoothed
plt.figure(7)
fd[10].scatter(s=0.5)
fd_us.plot(c='sandybrown')

# Over-smoothed
plt.figure(8)
fd[10].scatter(s=0.5)
fd_os.plot(c='r')
plt.show()
```

B.4 From discrete data to a basis representation

This script generates the graphs of the examples shown in Sections 2.3.1 and 2.3.2.

```
import fda
import numpy as np
import matplotlib.pyplot as plt

from fda import basis
from pathlib import Path


if __name__ == '__main__':
    # Loading dataset
    data = np.genfromtxt(Path('../data/phoneme_data.csv'),
                         delimiter=',',
                         skip_header=1)
    fd = fda.FDataGrid(data, list(range(data.shape[1])),
                       dataset_label='Phoneme learn',
                       axes_labels=['frequencies', 'log-periodograms'])

    # Least-squares example. Different number of basis
    fd[10].scatter(s=2, c='purple')
    fd[10].to_basis(basis.BSpline(nbasis=7, domain_range=fd.sample_range[0])
                    ).plot()

    fd[10].to_basis(basis.BSpline(nbasis=15, domain_range=fd.sample_range[0])
                    ).plot()

    fd[10].to_basis(basis.BSpline(nbasis=30, domain_range=fd.sample_range[0])
                    ).plot()

    plt.legend(['B-splines (7 basis)', 'B-splines (15 basis)',
                'B-splines (30 basis)', 'Original data'])

    # Roughness penalty example.
    plt.figure()
    fd[10].scatter(s=2, c='purple')
    fd[10].to_basis(basis.BSpline(nbasis=50, domain_range=fd.sample_range[
        0])).plot()
    fd[10].to_basis(basis.BSpline(nbasis=50, domain_range=fd.sample_range[
        0]), smoothness_parameter=100).plot(c='r')
    plt.legend(['B-splines (no penalty)', r'B-splines (roughness penalty) '
                                          r'$\lambda = 100$',
                'Original data'])
    plt.show()
```

C

Documentation

fda Documentation

Release 0.1

Miguel Carbajo Berrocal

May 06, 2018

Contents:

CHAPTER 1

fda package

1.1 fda.basis module

This module defines functional data object in a basis function system representation and the corresponding basis class.

class fda.basis.**BSpline**(*domain_range=None*, *nbasis=None*, *order=4*, *knots=None*)

Bases: *fda.basis.Basis*

BSpline basis.

BSpline basis elements are defined recursively as:

$$B_{i,1}(x) = 1 \quad \text{if } t_i \leq x < t_{i+1}, \quad 0 \text{ otherwise}$$

$$B_{i,k}(x) = \frac{x - t_i}{t_{i+k} - t_i} B_{i,k-1}(x) + \frac{t_{i+k+1} - x}{t_{i+k+1} - t_{i+1}} B_{i+1,k-1}(x)$$

Where k indicates the order of the spline.

Implementation details: In order to allow a discontinuous behaviour at the boundaries of the domain it is necessary to placing m knots at the boundaries *[RS05]*. This is automatically done so that the user only has to specify a single knot at the boundaries.

domain_range
tuple – A tuple of length 2 containing the initial and end values of the interval over which the basis can be evaluated.

nbasis
int – Number of functions in the basis.

order
int – Order of the splines. One greather than their degree.

knots
list – List of knots of the spline functions.

Examples

Constructs specifying number of basis and order.

```
>>> bss = BSpline(nbasis=8, order=4)
```

If no order is specified defaults to 4 because cubic splines are the most used. So the previous example is the same as:

```
>>> bss = BSpline(nbasis=8)
```

It is also possible to create a BSpline basis specifying the knots.

```
>>> bss = BSpline(knots=[0, 0.2, 0.4, 0.6, 0.8, 1])
```

Once we create a basis we can evaluate each of its functions at a set of points.

```
>>> bss = BSpline(nbasis=3, order=3)
>>> bss.evaluate([0, 0.5, 1])
array([[1.  , 0.25, 0.  ],
       [0.  , 0.5 , 0.  ],
       [0.  , 0.25, 1.  ]])
```

And evaluates first derivative

```
>>> bss.evaluate([0, 0.5, 1], derivative=1)
array([[-2., -1.,  0.],
       [ 2.,  0., -2.],
       [ 0.,  1.,  2.]])
```

References

penalty(*derivative_degree=None*, *coefficients=None*)

Returns a penalty matrix given a differential operator.

The differential operator can be either a derivative of a certain degree or a more complex operator.

The penalty matrix is defined as *[RS05-5-6-2]*:

$$R_{ij} = \int L\phi_i(s)L\phi_j(s)ds$$

where $\phi_i(s)$ for $i = 1, 2, ..., n$ are the basis functions and L is a differential operator.

Parameters

- **derivative_degree** (*int*) – Integer indicating the order of the derivative or . For instance 2 means that the differential operator is $f''(x)$.
- **coefficients** (*list*) – List of coefficients representing a differential operator. An iterable indicating coefficients of derivatives (which can be functions). For instance the tuple (1, 0, numpy.sin) means $1 + sin(x)D^2$. Only used if derivative degree is None.

Returns Penalty matrix.

Return type numpy.array

References

class fda.basis.**Basis**(*domain_range=(0, 1)*, *nbasis=1*)

Bases: abc.ABC

Defines the structure of a basis function system.

domain_range

tuple – a tuple of length 2 containing the initial and end values of the interval over which the basis can be evaluated.

nbasis

int – number of functions in the basis.

evaluate(*eval_points*, *derivative=0*)

Evaluates the basis function system or its derivatives at a list of given values.

Parameters

- **eval_points** (*array_like*) – List of points where the basis is evaluated.
- **derivative** (*int, optional*) – Order of the derivative. Defaults to 0.

Returns Matrix whose rows are the values of the each basis function or its derivatives at the values specified in eval_points.

Return type (numpy.darray)

penalty(*derivative_degree=None*, *coefficients=None*)

Returns a penalty matrix given a differential operator.

The differential operator can be either a derivative of a certain degree or a more complex operator.

The penalty matrix is defined as *[RS05-5-6-2]*:

$$R_{ij} = \int L\phi_i(s)L\phi_j(s)ds$$

where $\phi_i(s)$ for $i = 1, 2, ..., n$ are the basis functions and L is a differential operator.

Parameters

- **derivative_degree** (*int*) – Integer indicating the order of the derivative or . For instance 2 means that the differential operator is $f''(x)$.
- **coefficients** (*list*) – List of coefficients representing a differential operator. An iterable indicating coefficients of derivatives (which can be functions). For instance the tuple (1, 0, numpy.sin) means $1 + sin(x)D^2$. Only used if derivative degree is None.

Returns Penalty matrix.

Return type numpy.array

References

plot(*ax=None*, *derivative=0*, ***kwargs*)

Plots the basis object or its derivatives.

Parameters

- **ax** (*axis object, optional*) – axis over with the graphs are plotted. Defaults to matplotlib current axis.

- **derivative** (*int, optional*) – Order of the derivative. Defaults to 0.
- ****kwargs** – keyword arguments to be [RS05]_passed to the matplotlib.pyplot.plot function.

Returns List of lines that were added to the plot.

class fda.basis.**FDataBasis**(*basis*, *coefficients*)

Bases: object

Basis representation of functional data.

Class representation for functional data in the form of a set of basis functions multplied by a set of coefficients.

$$f(x) = \sum_{k=1} K c_k \phi_k$$

Where n is the number of basis functions, $c = (c_1, c_2, ..., c_K)$ the vector of coefficients and $\phi = (\phi_1, \phi_2, ..., \phi_K)$ the basis function system.

basis

Basis – Basis function system.

coefficients

numpy.darray – List or matrix of coefficients. Has to have the same length or number of columns as the number of basis function in the basis. If a matrix, each row contains the coefficients that multiplied by the basis functions produce each functional datum.

Examples

```
>>> basis = Monomial(nbasis=4)
>>> coefficients = [1, 1, 3, .5]
>>> FDataBasis(basis, coefficients)
FDataBasis(basis=Monomial(...), coefficients=[[1.  1.  3.  0.5]])
```

cov(*eval_points=None*)

Computes the covariance of the functional data object.

A numerical approach its used. The object its transformed into its discrete representation and then the covariance matrix is computed.

Parameters **eval_points** (*array_like, optional*) – Set of points where the functions are evaluated to obtain the discrete representation of the object. If none are passed it calls numpy.linspace with bounds equal to the ones defined in self.domain_range and the number of points the maximum between 501 and 10 times the number of basis.

Returns Matrix of covariances.

Return type numpy.darray

domain_range

Definition range

evaluate(*eval_points*, *derivative=0*)

Evaluates the object or its derivatives at a list of values.

Parameters

- **eval_points** (*array_like*) – List of points where the functions are evaluated.
- **derivative** (*int, optional*) – Order of the derivative. Defaults to 0.

Returns Matrix whose rows are the values of the each function at the values specified in eval_points.

Return type (numpy.darray)

classmethod from_data(*data_matrix*, *sample_points*, *basis*, *weight_matrix=None*, *smoothness_parameter=0*, *penalty_degree=None*, *penalty_coefficients=None*, *penalty_matrix=None*, *method='cholesky'*)

Raw data to a smooth functional form.

Takes functional data in a discrete form and makes an approximates it to the closest function that can be generated by the basis.a

The fit is made so as to reduce the penalized sum of squared errors *[RS05-5-2-5]*: .. math:

```
PENSSE(c) = (y - \Phi c)' W (y - \Phi c) + \lambda c'Rc
```

where y is the vector or matrix of observations, Φ the matrix whose columns are the basis functions evaluated at the sampling points, c the coefficient vector or matrix to be estimated, λ a smoothness parameter and $c'Rc$ the matrix representation of the roughness penalty $\int [L(x(s))]^2 \, ds$ where L is a linear differential operator.

Each element of R has the following close form: .. math:

```
R_{ij} = \int L\phi_i(s) L\phi_j(s) ds
```

By deriving the first formula we obtain the closed formed of the estimated coefficients matrix: .. math:

```
\hat(c) = \left( |Phi' W \Phi + \lambda R \right)^{-1} \Phi' W y
```

The solution of this matrix equation is done using the cholesky method for the resolution of a LS problem. If this method throughs a rounding error warning you may want to use the QR factorisation that is more numerically stable despite being more expensive to compute. *[RS05-5-2-7]*

Parameters

- **data_matrix** (*array_like*) – List or matrix containing the observations. If a matrix each row represents a single functional datum and the columns the different observations.
- **sample_points** (*array_like*) – Values of the domain where the previous data were taken.
- **basis** – (Basis): Basis used.
- **weight_matrix** (*array_like, optional*) – Matrix to weight the observations. Defaults to the identity matrix.
- **smoothness_parameter** (*int or float, optional*) – Smoothness parameter. Trying with several factors in a logarythm scale is suggested. If 0 no smoothing is performed. Defaults to 0.
- **penalty_degree** (*int*) – Integer indicating the order of the derivative used in the computing of the penalty matrix. For instance 2 means that the differential operator is $f''(x)$. If neither penalty_degree nor penalty_coefficients are supplied, this defaults to 2.
- **penalty_coefficients** (*list*) – List of coefficients representing the differential operator used in the computing of the penalty matrix. An iterable indicating coefficients of derivatives (which can be functions). For instance the tuple (1, 0, numpy.sin) means $1 + sin(x)D^2$. Only used if penalty_degree and penalty_matrix are None.
- **penalty_matrix** (*array_like, optional*) – Penalty matrix. If supplied the differential operator is not used and instead the matrix supplied by this argument is used.

- **method** (*str*) – Algorithm used for calculating the coefficients using the least squares method. The values admitted are 'cholesky' and 'qr' for Cholesky and QR factorisation methods respectively.

Returns

Represention of the data in a functional form as product of coefficients by basis functions.

Return type *FDataBasis*

Examples

```
>>> import numpy as np
>>> t = np.linspace(0, 1, 5)
>>> x = np.sin(2 * np.pi * t) + np.cos(2 * np.pi * t)
>>> x
array([ 1.,  1., -1., -1.,  1.])
```

```
>>> basis = Fourier((0, 1), nbasis=3)
>>> fd = FDataBasis.from_data(x, t, basis)
>>> fd.coefficients.round(2)
array([[0.  , 0.71, 0.71]])
```

References

gmean(*eval_points=None*)

Computes the geometric mean of the functional data object.

A numerical approach its used. The object its transformed into its discrete representation and then the geometric mean is computed and then the object is taken back to the basis representation.

Parameters **eval_points** (*array_like, optional*) – Set of points where the functions are evaluated to obtain the discrete representation of the object. If none are passed it calls numpy.linspace with bounds equal to the ones defined in self.domain_range and the number of points the maximum between 501 and 10 times the number of basis.

Returns Geometric mean of the original object.

Return type *FDataBasis*

mean()

Computes the mean of all the samples in a FDataBasis object.

Returns A FDataBais object with just one sample representing the mean of all the samples in the original FDataBasis object.

Return type *FDataBasis*

Examples

```
>>> basis = Monomial(nbasis=4)
>>> coefficients = [[0.5, 1, 2, .5], [1.5, 1, 4, .5]]
>>> FDataBasis(basis, coefficients).mean()
FDataBasis(basis=..., coefficients=[[1.  1.  3.  0.5]])
```

nbasis
Number of basis

nsamples
Number of samples

plot (*ax=None*, *derivative=0*, ***kwargs*)
Plots the FDataBasis object or its derivatives.

Parameters

- **ax** (*axis object, optional*) – axis over with the graphs are plotted. Defaults to matplotlib current axis.
- **derivative** (*int, optional*) – Order of the derivative. Default to 0.
- ****kwargs** – keyword arguments to be passed to the matplotlib.pyplot.plot function.

Returns List of lines that were added to the plot.

to_grid (*eval_points=None*)
Returns the discrete representation of the object.

Parameters **eval_points** (*array_like, optional*) – Set of points where the functions are evaluated. If none are passed it calls numpy.linspace with bounds equal to the ones defined in self.domain_range and the number of points the maximum between 501 and 10 times the number of basis.

Returns Discrete representation of the functional data object.

Return type *FDataGrid*

Examples

```
>>> fd = FDataBasis(coefficients=[[1, 1, 1], [1, 0, 1]],
...                 basis=Monomial((0,5), nbasis=3))
>>> fd.to_grid([0, 1, 2])
FDataGrid(
    array([[[1.],
            [3.],
            [7.]],

           [[1.],
            [2.],
            [5.]]]),
    sample_points=[array([0, 1, 2])],
    sample_range=array([[0, 5]]),
    dataset_label='Data set',
    axes_labels=None)
```

var (*eval_points=None*)
Computes the variance of the functional data object.

A numerical approach its used. The object its transformed into its discrete representation and then the variance is computed and then the object is taken back to the basis representation.

Parameters **eval_points** (*array_like, optional*) – Set of points where the functions are evaluated to obtain the discrete representation of the object. If none are passed it calls numpy.linspace with bounds equal to the ones defined in self.domain_range and the number of points the maximum between 501 and 10 times the number of basis.

Returns Variance of the original object.

Return type *FDataBasis*

class fda.basis.**Fourier**(*domain_range=(0, 1)*, *nbasis=3*, *period=1*)

Bases: *fda.basis.Basis*

Fourier basis.

Defines a functional basis for representing functions on a fourier series expansion of period T. The number of basis is always odd. If instantiated with an even number of basis, they will be incremented automatically by one.

$$\phi_0(t) = \frac{1}{\sqrt{2}}$$

$$\phi_{2n-1}(t) = sin\left(\frac{2\pi n}{T}t\right)$$

$$\phi_{2n}(t) = cos\left(\frac{2\pi n}{T}t\right)$$

Actually this basis functions are not orthogonal but not orthonormal. To achieve this they are divided by its norm: $\sqrt{\frac{T}{2}}$.

domain_range
tuple – A tuple of length 2 containing the initial and end values of the interval over which the basis can be evaluated.

nbasis
int – Number of functions in the basis.

period
int or float – Period (T).

Examples

Constructs specifying number of basis, definition interval and period.

```
>>> fb = Fourier([0, numpy.pi], nbasis=3, period=1)
>>> fb.evaluate([0, numpy.pi / 4, numpy.pi / 2, numpy.pi]).round(2)
array([[ 1.  ,  1.  ,  1.  ,  1.  ],
       [ 1.41,  0.31, -1.28,  0.89],
       [ 0.  , -1.38, -0.61,  1.1 ]])
```

And evaluate second derivative

```
>>> fb.evaluate([0, numpy.pi / 4, numpy.pi / 2, numpy.pi],
...             derivative = 2).round(2)
array([[  0.  ,   0.  ,   0.  ,   0.  ],
       [-55.83, -12.32,  50.4 , -35.16],
       [ -0.  ,  54.46,  24.02, -43.37]])
```

penalty(*derivative_degree=None*, *coefficients=None*)

Returns a penalty matrix given a differential operator.

The differential operator can be either a derivative of a certain degree or a more complex operator.

The penalty matrix is defined as *[RS05-5-6-2]*:

$$R_{ij} = \int L\phi_i(s)L\phi_j(s)ds$$

where $\phi_i(s)$ for $i = 1, 2, ..., n$ are the basis functions and L is a differential operator.

Parameters

- **derivative_degree** (*int*) – Integer indicating the order of the derivative or . For instance 2 means that the differential operator is $f''(x)$.
- **coefficients** (*list*) – List of coefficients representing a differential operator. An iterable indicating coefficients of derivatives (which can be functions). For instance the tuple (1, 0, numpy.sin) means $1 + sin(x)D^2$. Only used if derivative degree is None.

Returns Penalty matrix.

Return type numpy.array

References

class fda.basis.**Monomial**(*domain_range=(0, 1)*, *nbasis=1*)

Bases: *fda.basis.Basis*

Monomial basis.

Basis formed by powers of the argument t:

$$1, t, t^2, t^3 ...$$

domain_range
tuple – a tuple of length 2 containing the initial and end values of the interval over which the basis can be evaluated.

nbasis
int – number of functions in the basis.

Examples

Defines a monomial base over the interval $[0, 5]$ consisting on the first 3 powers of t: $1, t, t^2$.

```
>>> bs_mon = Monomial((0,5), nbasis=3)
```

And evaluates all the functions in the basis in a list of descrete values.

```
>>> bs_mon.evaluate([0, 1, 2])
array([[1., 1., 1.],
       [0., 1., 2.],
       [0., 1., 4.]])
```

And also evaluates its derivatives

```
>>> bs_mon.evaluate([0, 1, 2], derivative=1)
array([[0., 0., 0.],
       [1., 1., 1.],
       [0., 2., 4.]])
>>> bs_mon.evaluate([0, 1, 2], derivative=2)
array([[0., 0., 0.],
       [0., 0., 0.],
       [2., 2., 2.]])
```

penalty(*derivative_degree=None*, *coefficients=None*)

Returns a penalty matrix given a differential operator.

The differential operator can be either a derivative of a certain degree or a more complex operator.

The penalty matrix is defined as *[RS05-5-6-2]*:

$$R_{ij} = \int L\phi_i(s)L\phi_j(s)ds$$

where $\phi_i(s)$ for $i = 1, 2, ..., n$ are the basis functions and L is a differential operator.

Parameters

- **derivative_degree** (*int*) – Integer indicating the order of the derivative or . For instance 2 means that the differential operator is $f''(x)$.
- **coefficients** (*list*) – List of coefficients representing a differential operator. An iterable indicating coefficients of derivatives (which can be functions). For instance the tuple (1, 0, numpy.sin) means $1 + sin(x)D^2$. Only used if derivative degree is None.

Returns Penalty matrix.

Return type numpy.array

Examples

```
>>> Monomial(nbasis=4).penalty(2)
array([[ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  4.,  6.],
       [ 0.,  0.,  6., 12.]])
```

References

1.2 fda.grid module

Discretised functional data module.

This module defines a class for representing functional data as a series of lists of values, each representing the observation of a function measured in a list of discretisation points.

class fda.grid.**FDataGrid**(*data_matrix*, *sample_points=None*, *sample_range=None*, *dataset_label='Data set'*, *axes_labels=None*)

Bases: object

Represents discretised functional data.

Class for representing functional data as a set of curves discretised in a grid of points.

data_matrix

numpy.ndarray – a matrix where each entry of the first axis contains the values of a functional datum evaluated at the points of discretisation.

sample_points

numpy.ndarray – 2 dimension matrix where each row contains the points of dicretisation for each axis of data_matrix.

sample_range
numpy.ndarray – 2 dimension matrix where each row contains the bounds of the interval in which the functional data is considered to exist for each one of the axies.

dataset_label
str – name of the dataset.

axes_labels
list – list containing the labels of the different axis. The first element is the x label, the second the y label and so on.

Examples

Representation of a functional data object with 2 samples representing a function $f : \mathbb{R} \longmapsto \mathbb{R}$.

```
>>> data_matrix = [[1, 2], [2, 3]]
>>> sample_points = [2, 4]
>>> FDataGrid(data_matrix, sample_points)
FDataGrid(
    array([[[1],
            [2]],

           [[2],
            [3]]]),
    sample_points=[array([2, 4])],
    ...)
```

The number of columns of data_matrix have to be the length of sample_points.

```
>>> FDataGrid(numpy.array([1,2,4,5,8]), range(6))
Traceback (most recent call last):
    ....
ValueError: Incorrect dimension in data_matrix and sample_points...
```

FDataGrid support higher dimensional data both in the domain and image. Representation of a functional data object with 2 samples representing a function $f : \mathbb{R} \longmapsto \mathbb{R}^2$.

```
>>> data_matrix = [[[1, 0.3], [2, 0.4]], [[2, 0.5], [3, 0.6]]]
>>> sample_points = [2, 4]
>>> fd = FDataGrid(data_matrix, sample_points)
>>> fd.ndim_domain, fd.ndim_image
(1, 2)
```

Representation of a functional data object with 2 samples representing a function $f : \mathbb{R}^2 \longmapsto \mathbb{R}$. >>> data_matrix = [[[1, 0.3], [2, 0.4]], [[2, 0.5], [3, 0.6]]] >>> sample_points = [[2, 4], [3,6]] >>> fd = FDataGrid(data_matrix, sample_points) >>> fd.ndim_domain, fd.ndim_image (2, 1)

concatenate(*other*)
Joins samples from a similar FDataGrid object.

Joins samples from another FDataGrid object if it has the same dimensions and sampling points.

Parameters **other** (*FDataGrid*) – another FDataGrid object.

Returns FDataGrid object with the samples from the two original objects.

Return type *FDataGrid*

Examples

```
>>> fd = FDataGrid([1,2,4,5,8], range(5))
>>> fd_2 = FDataGrid([3,4,7,9,2], range(5))
>>> fd.concatenate(fd_2)
FDataGrid(
    array([[[1],
            [2],
            [4],
            [5],
            [8]],

           [[3],
            [4],
            [7],
            [9],
            [2]]]),
    sample_points=[array([0, 1, 2, 3, 4])],
    ...
```

cov()

Calculates the covariance matrix representing the covariance of the functional samples at the observation points.

Returns Matrix of covariances.

Return type numpy.darray

derivative(*order=1*)

Derivative of a FDataGrid object.

Its calculated using lagged differences. If we call D the data_matrix, D^1 the derivative of order 1 and T the vector contaning the points of discretisation; D^1 is calculated as it follows:

$$D^1_{ij} = \begin{cases} \frac{D_{i1}-D_{i2}}{T_1-T_2} & \text{if } j = 1 \\ \frac{D_{i(m-1)}-D_{im}}{T_{m-1}-T_m} & \text{if } j = m \\ \frac{D_{i(j-1)}-D_{i(j+1)}}{T_{j-1}-T_{j+1}} & \text{if } 1 < j < m \end{cases}$$

Where m is the number of columns of the matrix D.

Order > 1 derivatives are calculated by using derivative recursively.

Parameters **order** (*int, optional*) – Order of the derivative. Defaults to one.

Examples

First order derivative

```
>>> fdata = FDataGrid([1,2,4,5,8], range(5))
>>> fdata.derivative()
FDataGrid(
    array([[[1. ],
            [1.5],
            [1.5],
            [2. ],
            [3. ]]]),
    sample_points=[array([0, 1, 2, 3, 4])],
```

```
    sample_range=array([[0, 4]]),
    dataset_label='Data set - 1 derivative',
    ...)
```

Second order derivative

```
>>> fdata = FDataGrid([1,2,4,5,8], range(5))
>>> fdata.derivative(2)
FDataGrid(
    array([[[0.5 ],
            [0.25],
            [0.25],
            [0.75],
            [1.  ]]]),
    sample_points=[array([0, 1, 2, 3, 4])],
    sample_range=array([[0, 4]]),
    dataset_label='Data set - 2 derivative',
    ...)
```

gmean()
Computes the geometric mean of all samples in the FDataGrid object.

Returns A FDataGrid object with just one sample representing the geometric mean of all the samples in the original FDataGrid object.

Return type *FDataGrid*

mean()

ncol
Number of columns of the data_matrix. Also the number of points of discretisation.

Returns Number of columns of the data_matrix.

Return type int

ndim
Number of dimensions of the data matrix.

Returns Number of dimensions of the data matrix.

Return type int

ndim_domain
Number of dimensions of the domain.

Returns Number of dimensions of the domain.

Return type int

ndim_image
Number of dimensions of the image

Returns Number of dimensions of the image.

Return type int

nsamples
Number of rows of the data_matrix. Also the number of samples.

Returns

Number of samples of the FDataGrid object. Also the number of rows of the data_matrix.

Return type int

plot (*ax=None*, ***kwargs*)
Plots the FDatGrid object.

Parameters

- **ax** (*axis object, optional*) – axis over with the graphs are plotted. Defaults to matplotlib current axis.
- ****kwargs** – keyword arguments to be passed to the matplotlib.pyplot.plot function.

Returns List of lines that were added to the plot.

round (*decimals=0*)
Evenly round to the given number of decimals.

Parameters **decimals** (*int, optional*) – Number of decimal places to round to. If decimals is negative, it specifies the number of positions to the left of the decimal point. Defaults to 0.

Returns obj:FDataGrid: Returns a FDataGrid object where all elements in its data_matrix are rounded .The real and imaginary parts of complex numbers are rounded separately.

scatter (*ax=None*, ***kwargs*)
Scatter plot of the FDatGrid object.

Parameters

- **ax** (*axis object, optional*) – axis over with the graphs are plotted. Defaults to matplotlib current axis.
- ****kwargs** – keyword arguments to be passed to the matplotlib.pyplot.scatter function.

Returns `matplotlib.collections.PathCollection`

shape
Dimensions (aka shape) of the data_matrix.

Returns List containing the length of the matrix on each of its axis. If the matrix is 2 dimensional shape returns [number of rows, number of columns].

Return type list of int

to_basis (*basis*, ***kwargs*)
Returns the basis representation of the object.

Parameters

- **basis** (Basis) – basis object in which the functional data are going to be represented.
- ****kwargs** – keyword arguments to be passed to FDataBasis.from_data().

Returns Basis representation of the funtional data object.

Return type *FDataBasis*

Examples

```
>>> import numpy as np
>>> import fda
>>> t = np.linspace(0, 1, 5)
>>> x = np.sin(2 * np.pi * t) + np.cos(2 * np.pi * t)
```

```
>>> x
array([ 1.,  1., -1., -1.,  1.])
```

```
>>> fd = FDataGrid(x, t)
>>> basis = fda.basis.Fourier((0, 1), nbasis=3)
>>> fd_b = fd.to_basis(basis)
>>> fd_b.coefficients.round(2)
array([[0.  , 0.71, 0.71]])
```

var()

Computes the variance of a set of samples in a FDataGrid object.

Returns A FDataGrid object with just one sample representing the variance of all the samples in the original FDataGrid object.

Return type *FDataGrid*

1.3 fda.kernel_smoothers module

Kernel smoother functions

This module includes the most commonly used kernel smoother methods for FDA. So far only non parametric methods are implemented because we are only relaying on a discrete representation of functional data.

Todo:

- Closed-form for KNN

fda.kernel_smoothers.**knn**(*argvals*, *k=None*, *kernel=<function uniform>*, *w=None*, *cv=False*)

K-nearest neighbour kernel smoother.

Provides an smoothing matrix S for the discretisation points in argvals by the k nearest neighbours estimator.

Usually used with the uniform kernel, it takes the average of the closest k points to a given point.

Parameters

- **argvals** (*ndarray*) – Vector of discretisation points.
- **k** (*int, optional*) – Number of nearest neighbours. By default it takes the 5% closest points.
- **kernel** (*function, optional*) – kernel function. By default a uniform kernel to perform a 'usual' k nearest neighbours estimation.
- **w** (*ndarray, optional*) – Case weights matrix.
- **cv** (*bool, optional*) – Flag for cross-validation methods. Defaults to False.

Returns Smoothing matrix.

Return type ndarray

Examples

```
>>> knn(numpy.array([1,2,4,5,7]), 2)
array([[0.5, 0.5, 0. , 0. , 0. ],
       [0.5, 0.5, 0. , 0. , 0. ],
       [0. , 0. , 0.5, 0.5, 0. ],
       [0. , 0. , 0.5, 0.5, 0. ],
       [0. , 0. , 0. , 0.5, 0.5]])
```

In case there are two points at the same distance it will take both.

```
>>> knn(numpy.array([1,2,3,5,7]), 2).round(3)
array([[0.5  , 0.5  , 0.   , 0.   , 0.   ],
       [0.333, 0.333, 0.333, 0.   , 0.   ],
       [0.   , 0.5  , 0.5  , 0.   , 0.   ],
       [0.   , 0.   , 0.333, 0.333, 0.333],
       [0.   , 0.   , 0.   , 0.5  , 0.5  ]])
```

fda.kernel_smoothers.**local_linear_regression**(*argvals*, *h*, *kernel=<function normal>*, *w=None*, *cv=False*)

Local linear regression smoothing method.

Provides an smoothing matrix $\hat{H}$ for the discretisation points in argvals by the local linear regression estimator. The smoothed values $\hat{Y}$ can be calculated as $\hat{Y} = \hat{H}Y$ where Y is the vector of observations at the points of discretisation $(x_1, x_2, ..., x_n)$.

$$\hat{H}_{i,j} = \frac{b_i(x_j)}{\sum_{k=1}^{n} b_k(x_j)}$$

$$b_i(x) = K\left(\frac{x_i - x}{h}\right) S_{n,2}(x) - (x_i - x) S_{n,1}(x)$$

$$S_{n,k} = \sum_{i=1}^{n} K\left(\frac{x_i - x}{h}\right) (x_i - x)^k$$

where $K(\cdot)$ is a kernel function and h the kernel window width.

Parameters

- **argvals** (*ndarray*) – Vector of discretisation points.
- **h** (*float, optional*) – Window width of the kernel.
- **kernel** (*function, optional*) – kernel function. By default a normal kernel.
- **w** (*ndarray, optional*) – Case weights matrix.
- **cv** (*bool, optional*) – Flag for cross-validation methods. Defaults to False.

Examples

```
>>> local_linear_regression(numpy.array([1,2,4,5,7]), 3.5).round(3)
array([[ 0.614,  0.429,  0.077, -0.03 , -0.09 ],
       [ 0.381,  0.595,  0.168, -0.   , -0.143],
       [-0.104,  0.112,  0.697,  0.398, -0.104],
       [-0.147, -0.036,  0.392,  0.639,  0.152],
       [-0.095, -0.079,  0.117,  0.308,  0.75 ]])
>>> local_linear_regression(numpy.array([1,2,4,5,7]), 2).round(3)
array([[ 0.714,  0.386, -0.037, -0.053, -0.01 ],
       [ 0.352,  0.724,  0.045, -0.081, -0.04 ],
       [-0.078,  0.052,  0.74 ,  0.364, -0.078],
```

```
       [-0.07 , -0.067,  0.36 ,  0.716,  0.061],
       [-0.012, -0.032, -0.025,  0.154,  0.915]])
```

Returns Smoothing matrix $\hat{H}$.

Return type ndarray

fda.kernel_smoothers.**nw**(*argvals*, *h=None*, *kernel=<function normal>*, *w=None*, *cv=False*)
Nadaraya-Watson smoothing method.

Provides an smoothing matrix $\hat{H}$ for the discretisation points in argvals by the Nadaraya-Watson estimator. The smoothed values $\hat{Y}$ can be calculated as $\hat{Y} = \hat{H}Y$ where Y is the vector of observations at the points of discretisation $(x_1, x_2, ..., x_n)$.

$$\hat{H}_{i,j} = \frac{K\left(\frac{x_i - x_j}{h}\right)}{\sum_{k=1}^{n} K\left(\frac{x_1 - x_k}{h}\right)}$$

where $K(\cdot)$ is a kernel function and h the kernel window width.

Parameters

- **argvals** (*ndarray*) – Vector of discretisation points.
- **h** (*float, optional*) – Window width of the kernel.
- **kernel** (*function, optional*) – kernel function. By default a normal kernel.
- **w** (*ndarray, optional*) – Case weights matrix.
- **cv** (*bool, optional*) – Flag for cross-validation methods. Defaults to False.

Examples

```
>>> nw(numpy.array([1,2,4,5,7]), 3.5).round(3)
array([[0.294, 0.282, 0.204, 0.153, 0.068],
       [0.249, 0.259, 0.22 , 0.179, 0.093],
       [0.165, 0.202, 0.238, 0.229, 0.165],
       [0.129, 0.172, 0.239, 0.249, 0.211],
       [0.073, 0.115, 0.221, 0.271, 0.319]])
>>> nw(numpy.array([1,2,4,5,7]), 2).round(3)
array([[0.425, 0.375, 0.138, 0.058, 0.005],
       [0.309, 0.35 , 0.212, 0.114, 0.015],
       [0.103, 0.193, 0.319, 0.281, 0.103],
       [0.046, 0.11 , 0.299, 0.339, 0.206],
       [0.006, 0.022, 0.163, 0.305, 0.503]])
```

Returns Smoothing matrix $\hat{H}$.

Return type ndarray

1.4 fda.kernels module

This module defines the most commonly used kernels.

fda.kernels.**cosine**(*u*)
Cosine kernel.

$$K(x) = \begin{cases} \frac{\pi}{4} cos\left(\frac{\pi x}{2}\right) & \text{if } |x| \leq 1 \\ 0 & \text{elsewhere} \end{cases}$$

fda.kernels.**epanechnikov**(*u*)
Epanechnikov kernel.

$$K(x) = \begin{cases} 0.75(1 - x^2) & \text{if } |x| \leq 1 \\ 0 & \text{elsewhere} \end{cases}$$

fda.kernels.**normal**(*u*)
Normal kernel.

$$K(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}$$

fda.kernels.**quartic**(*u*)
Quartic kernel.

$$K(x) = \begin{cases} \frac{15}{16}\left(1 - u^2\right)^2 & \text{if } |x| \leq 1 \\ 0 & \text{elsewhere} \end{cases}$$

fda.kernels.**tri_weight**(*u*)
Tri-weight kernel.

$$K(x) = \begin{cases} \frac{35}{32}\left(1 - u^2\right)^3 & \text{if } |x| \leq 1 \\ 0 & \text{elsewhere} \end{cases}$$

fda.kernels.**uniform**(*u*)
Uniform kernel.

$$K(x) = \begin{cases} 0.5 & \text{if } |x| \leq 1 \\ 0 & \text{elsewhere} \end{cases}$$

1.5 fda.math module

This module defines the basic mathematic operations for classes defined in this package.

fda.math.**absolute**(*fdatagrid*)
Gets the absolute value of all elements in the FDataGrid object.

Parameters **fdatagrid** (FDataGrid) – Object from whose elements the absolute value is going to be retrieved.

Returns

Object whose elements are the absolute values of the original.

Return type *FDataGrid*

fda.math.**cov**(*fdatagrid*)
Calculates the covariance matrix representing the covariance of the functional samples at the observation points.

Parameters **fdatagrid** (FDataGrid) – Object containing different samples of a functional variable.

Returns Matrix of covariances.

Return type numpy.darray

fda.math.**cumsum**(*fdatagrid*)
Returns the cumulative sum of the samples.

Parameters **fdatagrid** (FDataGrid) – Object over whose samples the cumulative sum is going to be calculated.

Returns Object with the sample wise cumulative sum.

Return type *FDataGrid*

fda.math.**exp**(*fdatagrid*)
Performs a element wise exponential operation.

Parameters **fdatagrid** (FDataGrid) – Object to whose elements the exponential operation is going to be applied.

Returns

Object whose elements are the result of exponentiating the elements of the original.

Return type *FDataGrid*

fda.math.**fpca**(*fdatagrid*, *n=2*)
Functional Principal Components Analysis.

Performs Functional Principal Components Analysis to reduce dimensionality and obtain the principal modes of variation for a functional data object.

It uses SVD numpy implementation to compute PCA.

Parameters

- **fdatagrid** (FDataGrid) – functional data object.
- **n** (*int, optional*) – Number of principal components. Defaults to 2.

Returns (scores, principal directions, eigenvalues)

Return type tuple

fda.math.**gmean**(*fdatagrid*)
Computes the geometric mean of all the samples in a FDataGrid object.

Parameters **fdatagrid** (FDataGrid) – Object containing all the samples whose geometric mean is wanted.

Returns A FDataGrid object with just one sample representing the geometric mean of all the samples in the original FDataGrid object.

Return type *FDataGrid*

fda.math.**inner_product**(*fdatagrid*, *fdatagrid2*)
Calculates the inner product amongst all the samples in two FDataGrid objects.

For each pair of samples f and g the inner product is defined as:

$$< f, g >= \int_a^b f(x)g(x)dx$$

The integral is approximated using Simpson's rule.

Parameters

- **fdatagrid** (FDataGrid) – First FDataGrid object.
- **fdatagrid2** (FDataGrid) – Second FDataGrid object.

Returns Matrix with as many rows as samples in the first object and as many columns as samples in the second one. Each element (i, j) of the matrix is the inner product of the ith sample of the first object and the jth sample of the second one.

Return type numpy.darray

Examples

The inner product of the :math:'f(x) = x' and the constant $y = 1$ defined over the interval [0,1] is the area of the triangle delimited by the the lines y = 0, x = 1 and y = x; 0.5.

```
>>> x = numpy.linspace(0,1,1001)
>>> fd1 = FDataGrid(x,x)
>>> fd2 = FDataGrid(numpy.ones(len(x)),x)
>>> inner_product(fd1, fd2)
array([[0.5]])
```

If the FDataGrid object contains more than one sample

```
>>> fd1 = FDataGrid([x, numpy.ones(len(x))], x)
>>> fd2 = FDataGrid([numpy.ones(len(x)), x] ,x)
>>> inner_product(fd1, fd2).round(2)
array([[0.5 , 0.33],
       [1.  , 0.5 ]])
```

fda.math.**log**(*fdatagrid*)
Performs a element wise logarithm operation.

Parameters **fdatagrid** (FDataGrid) – Object to whose elements the logarithm operation is going to be applied.

Returns Object whose elements are the logarithm of the original.

Return type *FDataGrid*

fda.math.**log10**(*fdatagrid*)
Performs a element wise base 10 logarithm operation.

Parameters **fdatagrid** (FDataGrid) – Object to whose elements the base 10 logarithm operation is going to be applied.

Returns

Object whose elements are the base 10 logarithm of the original.

Return type *FDataGrid*

fda.math.**log2**(*fdatagrid*)
Performs a element wise binary logarithm operation.

Parameters **fdatagrid** (FDataGrid) – Object to whose elements the binary logarithm operation is going to be applied.

Returns

Object whose elements are the binary logarithm of the original.

Return type *FDataGrid*

fda.math.**mean**(*fdata*)

Computes the mean of all the samples in a FData object.

Computes the mean of all the samples in a FDataGrid or FDataBasis object.

Parameters

- **fdata** (FDataGrid *or* FDataBasis) – Object containing all the samples
- **mean** (*whose*) – is wanted.

Returns A FDataGrid or FDataBasis object with just one sample representing the mean of all the samples in the original object.

Return type *FDataGrid* or *FDataBasis*

fda.math.**metric**(*fdatagrid*, *fdatagrid2*, *norm=<function norm_lp>*, ***kwargs*)

Calculates the distance between all possible pairs of one sample of the first FDataGrid object and one of the second one.

For each pair of samples f and g the distance between them is defined as:

$$d(f, g) = d(f, g) = \|f - g\|$$

The norm is specified as a parameter but defaults to the l2 norm.

Parameters

- **fdatagrid** (FDataGrid) – First FDataGrid object.
- **fdatagrid2** (FDataGrid) – Second FDataGrid object.
- **norm** (Function, optional) – Norm function used in the definition of the distance.
- ****kwargs** (dict, optional) – parameters dictionary to be passed to the norm function.

Returns Matrix with as many rows as samples in the first object and as many columns as samples in the second one. Each element (i, j) of the matrix is the distance between the ith sample of the first object and the jth sample of the second one.

Return type numpy.darray

Examples

Computes the distances between an object containing functional data corresponding to the functions y = 1 and y = x defined over the interval [0, 1] and another ones containing data of the functions y = 0 and y = x/2. The result then is an array 2x2 with the computed l2 distance between every pair of functions.

```
>>> x = numpy.linspace(0, 1, 1001)
>>> fd = FDataGrid([numpy.ones(len(x)), x], x)
>>> fd2 =  FDataGrid([numpy.zeros(len(x)), x/2 + 0.5], x)
>>> metric(fd, fd2).round(2)
array([[1.  , 0.29],
       [0.58, 0.29]])
```

If the functional data are defined over a different set of points of discretisation the functions returns an exception.

```
>>> x = numpy.linspace(0, 2, 1001)
>>> fd2 =  FDataGrid([numpy.zeros(len(x)), x/2 + 0.5], x)
>>> metric(fd, fd2)
```

```
Traceback (most recent call last):
    ....
ValueError: Sample points for both objects must be equal
```

fda.math.**norm_lp**(*fdatagrid*, *p=2*)

Calculates the norm of all the samples in a FDataGrid object.

For each sample sample f the lp norm is defined as:

$$\|f\| = \left(\int_D |f|^p dx\right)^{\frac{1}{p}}$$

Where D is the domain over which the functions are defined.

The integral is approximated using Simpson's rule.

Parameters

- **fdatagrid** (FDataGrid) – FDataGrid object.
- **p** (*int, optional*) – p of the lp norm. Must be greater or equal than 1. Defaults to 2.

Returns Matrix with as many rows as samples in the first object and as many columns as samples in the second one. Each element (i, j) of the matrix is the inner product of the ith sample of the first object and the jth sample of the second one.

Return type numpy.darray

Examples

Calculates the norm of a FDataGrid containing the functions y = 1 and y = x defined in the interval [0,1].

```
>>> x = numpy.linspace(0,1,1001)
>>> fd = FDataGrid([numpy.ones(len(x)), x] ,x)
>>> norm_lp(fd).round(2)
array([1.  , 0.58])
```

The lp norm is only defined if p >= 1.

```
>>> norm_lp(fd, p = 0.5)
Traceback (most recent call last):
    ....
ValueError: p must be equal or greater than 1.
```

fda.math.**round**(*fdatagrid*, *decimals=0*)

Rounds all elements of the object.

Parameters

- **fdatagrid** (FDataGrid) – Object to whose elements are going to be rounded.
- **decimals** (*int, optional*) – Number of decimals wanted. Defaults to 0.

Returns Object whose elements are rounded.

Return type *FDataGrid*

fda.math.**sqrt**(*fdatagrid*)

Performs a element wise square root operation.

Parameters **fdatagrid** (FDataGrid) – Object to whose elements the square root operation is going to be applied.

Returns Object whose elements are the square roots of the original.

Return type *FDataGrid*

fda.math.**var**(*fdatagrid*)

Computes the variance of a set of samples in a FDataGrid object.

Parameters

- **fdatagrid** (FDataGrid) – Object containing all the set of samples
- **variance is desired.** (*whose*) –

Returns A FDataGrid object with just one sample representing the variance of all the samples in the original FDataGrid object.

Return type *FDataGrid*

1.6 fda.validation module

This module defines methods for the validation of the smoothing.

fda.validation.**aic**(*s_matrix*)

Akaike's information criterion for cross validation.

$$\Xi(\nu, n) = \exp\left(2 * \frac{tr(\hat{H}^\nu)}{n}\right)$$

Parameters **s_matrix** (*numpy.darray*) – Smoothing matrix whose penalisation score is desired.

Returns Penalisation given by the Akaike's information criterion.

Return type float

fda.validation.**cv**(*fdatagrid*, *s_matrix*)

Cross validation scoring method.

It calculates the cross validation score for every sample in a FDataGrid object given a smoothing matrix $\hat{H}^\nu$ calculated with a parameter ν:

$$CV(\nu) = \frac{1}{n}\sum_i \left(y_i - \hat{y}_i^{\nu(-i)}\right)^2$$

Where $\hat{y}_i^{\nu(-i)}$ is the adjusted y_i when the the pair of values (x_i, y_i) are excluded in the smoothing. This would require to recalculate the smoothing matrix n times. Fortunately the above formula can be expressed in a way where the smoothing matrix does not need to be calculated again.

$$CV(\nu) = \frac{1}{n}\sum_i \left(\frac{y_i - \hat{y}_i^\nu}{1 - \hat{H}_{ii}^\nu}\right)^2$$

Parameters

- **fdatagrid** (FDataGrid) – Object over which the CV score is calculated.
- **s_matrix** (*numpy.darray*) – Smoothig matrix.

Returns Cross validation score.

Return type float

fda.validation.**fpe**(*s_matrix*)
Finite prediction error for cross validation.

$$\Xi(\nu, n) = \frac{1 + \frac{tr(\hat{H}^\nu)}{n}}{1 - \frac{tr(\hat{H}^\nu)}{n}}$$

Parameters **s_matrix** (*numpy.darray*) – Smoothing matrix whose penalisation score is desired.

Returns Penalisation given by the finite prediction error.

Return type float

fda.validation.**gcv**(*fdatagrid*, *s_matrix*, *penalisation_function=None*)
General cross validation scoring method.

It calculates the general cross validation score for every sample in a FDataGrid object given a smoothing matrix $\hat{H}^\nu$ calculated with a parameter ν:

$$GCV(\nu) = \Xi(\nu, n)\frac{1}{n}\sum_i (y_i - \hat{y}_i^\nu)^2$$

Where $\hat{y}_i^\nu$ is the adjusted y_i and Ξ is a penalisation function. By default the penalisation function is:

$$\Xi(\nu, n) = \left(1 - \frac{tr(\hat{H}^\nu)}{n}\right)^{-2}$$

But others such as the Akaike's information criterion can be considered.

Parameters

- **fdatagrid** (FDataGrid) – Object over which the CV score is calculated.
- **s_matrix** (*numpy.darray*) – Smoothig matrix.
- **penalisation_function** (*Function*) – Function taking a smoothing matrix and returing a penalisation score. If None the general cross validation penalisation is applied. Defaults to None.

Returns Cross validation score.

Return type float

fda.validation.**minimise**(*fdatagrid*, *parameters*, *smoothing_method=<function nw>*, *cv_method=<function gcv>*, *penalisation_function=None*, ***kwargs*)
Performs the smoothing of a FDataGrid object choosing the best parameter of a given list using a cross validation scoring method.

Parameters

- **fdatagrid** (FDataGrid) – FDataGrid object.
- **parameters** (*list of double*) – List of parameters to be tested.
- **smoothing_method** (*Function*) – Function that takes a list of discretised points, a parameter, an optionally a weights matrix and returns a hat matrix or smoothing matrix.
- **cv_method** (*Function*) – Function that takes a matrix, a smoothing matrix, and optionally a weights matrix and calculates a cross validation score.
- **penalisation_function** (*Fuction*) – if gcv is selected as cv_method a penalisation function can be specified through this parameter.

Returns

A dictionary containing the following:

{

'scores': (list of double) List of the scores for each parameter.

'best_score': (double) Minimum score.

'best_parameter': (double) Parameter that produces the lesser score.

'hat_matrix': (numpy.darray) Hat matrix built with the best parameter.

'fdatagrid': (FDataGrid) Smoothed FDataGrid object.

}

Return type dict

Examples

Creates a FDataGrid object of the function $y = x^2$ and peforms smoothing by means of the k-nearest neighbours method.

```
>>> x = numpy.linspace(-2, 2, 5)
>>> fd = fda.FDataGrid(x ** 2, x)
>>> res = minimise(fd, [2,3], smoothing_method=kernel_smoothers.knn)
>>> numpy.array(res['scores']).round(2)
array([11.67, 12.37])
>>> round(res['best_score'], 2)
11.67
>>> res['best_parameter']
2
>>> res['hat_matrix'].round(2)
array([[0.5 , 0.5 , 0.  , 0.  , 0.  ],
       [0.33, 0.33, 0.33, 0.  , 0.  ],
       [0.  , 0.33, 0.33, 0.33, 0.  ],
       [0.  , 0.  , 0.33, 0.33, 0.33],
       [0.  , 0.  , 0.  , 0.5 , 0.5 ]])
>>> res['fdatagrid'].round(2)
FDataGrid(
    array([[[2.5 ],
            [1.67],
            [0.67],
            [1.67],
            [2.5 ]]]),
    sample_points=[array([-2., -1.,  0.,  1.,  2.])],
    sample_range=array([[-2.,  2.]]),
    ...)
```

Other validation methods can be used such as cross-validation or general cross validation using other penalisation functions.

```
>>> res = minimise(fd, [2,3], smoothing_method=kernel_smoothers.knn,
...                cv_method=cv)
>>> numpy.array(res['scores']).round(2)
array([4.2, 5.5])
>>> res = minimise(fd, [2,3], smoothing_method=kernel_smoothers.knn,
...                penalisation_function=aic)
```

```
>>> numpy.array(res['scores']).round(2)
array([ 9.35, 10.71])
>>> res = minimise(fd, [2,3], smoothing_method=kernel_smoothers.knn,
...                penalisation_function=fpe)
>>> numpy.array(res['scores']).round(2)
array([ 9.8, 11. ])
>>> res = minimise(fd, [2,3], smoothing_method=kernel_smoothers.knn,
...                penalisation_function=shibata)
>>> numpy.array(res['scores']).round(2)
array([7.56, 9.17])
>>> res = minimise(fd, [2,3], smoothing_method=kernel_smoothers.knn,
...                penalisation_function=rice)
>>> numpy.array(res['scores']).round(2)
array([21. , 16.5])
```

fda.validation.**rice**(*s_matrix*)

Rice's bandwidth selector for cross validation.

$$\Xi(\nu, n) = \left(1 - 2 * \frac{tr(\hat{H}^{\nu})}{n}\right)^{-1}$$

Parameters **s_matrix** (*numpy.darray*) – Smoothing matrix whose penalisation score is desired.

Returns Penalisation given by the Rice's bandwidth selector.

Return type float

fda.validation.**shibata**(*s_matrix*)

Shibata's model selector for cross validation.

$$\Xi(\nu, n) = 1 + 2 * \frac{tr(\hat{H}^{\nu})}{n}$$

Parameters **s_matrix** (*numpy.darray*) – Smoothing matrix whose penalisation score is desired.

Returns Penalisation given by the Shibata's model selector.

Return type float

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Bibliography

[RS05] Ramsay, J., Silverman, B. W. (2005). *Functional Data Analysis*. Springler. 50-51.

[RS05-5-6-2] Ramsay, J., Silverman, B. W. (2005). Specifying the roughness penalty. In *Functional Data Analysis* (pp. 106-107). Springler.

[RS05-5-6-2] Ramsay, J., Silverman, B. W. (2005). Specifying the roughness penalty. In *Functional Data Analysis* (pp. 106-107). Springler.

[RS05-5-2-5] Ramsay, J., Silverman, B. W. (2005). How spline smooths are computed. In *Functional Data Analysis* (pp. 86-87). Springler.

[RS05-5-2-7] Ramsay, J., Silverman, B. W. (2005). HSpline smoothing as an augmented least squares problem. In *Functional Data Analysis* (pp. 86-87). Springler.

[RS05-5-6-2] Ramsay, J., Silverman, B. W. (2005). Specifying the roughness penalty. In *Functional Data Analysis* (pp. 106-107). Springler.

[RS05-5-6-2] Ramsay, J., Silverman, B. W. (2005). Specifying the roughness penalty. In *Functional Data Analysis* (pp. 106-107). Springler.

Python Module Index

f

Index

N

O

P

Q

R

S

T

U

V